Module-1

Searching: Linear and Binary Search

Program1:

Find First and Last Position of Element in Sorted Array(Leetcode 34):

Problem Statement:

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

Constraints:

- $0 <= nums.length <= 10^5$
- $-10^9 <= nums[i] <= 10^9$
- nums is a non-decreasing array.
- $-10^9 <= target <= 10^9$

Algorithm:

Step1: Takes a sorted array (nums) and a target value.

Step2: Uses two helper functions:

- first_occurance() — finds the **first index** of the target using modified binary search.

  Use Binary Search to find the first occurrence of the target:

  If found, store the index and continue searching in the left half.

- last_occurance() — finds the **last index** of the target similarly.

  Use Binary Search to find the last occurrence of the target:

  If found, store the index and continue searching in the right half.

Step3: Returns a vector with both indices.

Step4: If the target doesn't exist, both functions return -1.

Implementation:

```
#include<iostream>
#include<vector>
using namespace std;
int first_occurance(vector<int> &nums,int target)
{
    int l=0;
    int h=nums.size()-1;
    int ans=-1;
```

```
    while(l<=h)
    {
        int m=l+(h-l)/2;
        if(nums[m]==target)
        {
            ans=m;
            h=m-1;
        }
        else if(nums[m]<target)
        {
            l=m+1;
        }
        else{
            h=m-1;
        }
    }
    return ans;
}
int last_occurance(vector<int> &nums,int target)
{
    int l=0;
    int h=nums.size()-1;
    int ans=-1;
    while(l<=h)
    {
        int m=(l+h)/2;
        if(nums[m]==target)
        {
            ans=m;
            l=m+1;
        }
        else if(nums[m]<target)
        {
            l=m+1;
        }
        else{
            h=m-1;
        }

    }
    return ans;
}
vector<int> searchRange(vector<int>& nums, int target) {
        vector<int>ans;
        ans.push_back(first_occurance(nums,target));
        ans.push_back(last_occurance(nums,target));
        return ans;
}
int main()
{
    int n;
    cin>>n;
    vector<int>nums(n);
    for(int i=0;i<n;i++)
    {
        cin>>nums[i];
```

```
    }
    int target;
    cin>>target;
    vector<int> res=searchRange(nums,target);
    cout<<"[ ";
    for(auto i:res){
        cout<<i<<" ";
    }
    cout<<"]";
}
```

Expected Results:

Input: n=6 nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

Input: n=6 nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

Program2:

Problem Statement:

Search in Rotated Sorted Array(Leetcode 33):

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return *the index of* target *if it is in* nums*, or* -1 *if it is not in* nums.

You must write an algorithm with O(log n) runtime complexity.

Constraints:

- 1 <= nums.length <= 5000
- -10^4 <= nums[i] <= 10^4
- All values of nums are unique.
- nums is an ascending array that is possibly rotated.
- -10^4 <= target <= 10^4

Algorithm:

Step1. Initialize low = 0 and high = n-1.

Step2. Perform Binary Search:

- Compute mid = (low + high) / 2.
- If nums[mid] == target, return mid.
- Determine which half of the array is sorted:
  - If nums[low] <= nums[mid], the left half is sorted.
  - If target lies within this range, search in the left half.
  - Otherwise, search in the right half.
  - Else, the right half is sorted.
  - If target lies within this range, search in the right half.
  - Otherwise, search in the left half.

Step3: If the target is not found, return -1.

Implementation:

```
#include <iostream>
#include <vector>
using namespace std;
int searchInRotatedArray(vector<int>& nums, int target) {
int low = 0, high = nums.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target)
            return mid;
        if (nums[low] <= nums[mid]) {
            if (nums[low] <= target && target < nums[mid])
                high = mid - 1;
            else
                low = mid + 1;
        } else {
            if (nums[mid] < target && target <= nums[high])
                low = mid + 1;
            else
                high = mid - 1;
        }
    }
    return -1;
}
int main() {
int n, target;
cin >> n;
vector<int> nums(n);
for (int i = 0; i < n; i++) {
cin >> nums[i];
}
cin >> target;
int result = searchInRotatedArray(nums,target);
if (result != -1)
cout << "Element found at index " << result << endl;
else
cout << "Element not found." << endl;
return 0;
```

```
}
```

Expected Results:

Input: nums = [4,5,6,7,0,1,2], target = 0

Output: 4

Input: nums = [4,5,6,7,0,1,2], target = 3

Output: -1

Input: nums = [1], target = 0

Output: -1

Program3:

Problem Statement:

Find Peak Element (Leetcode 162):

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

Constraints:

- 1 <= nums.length <= 1000
- -2^{31} <= nums[i] <= 2^{31} - 1
- nums[i] != nums[i + 1] for all valid i

Algorithm:

Step1. Initialize low = 0 and high = n-1.

Step2. Perform Binary Search to find peak element:

- Compute mid = (low + high) / 2.
- If nums[mid] > nums[mid + 1], then mid is a peak or the peak is in the left half.
- Otherwise, the peak is in the right half.

Step3. Continue searching until the peak is found.

Implementation:

```
#include <iostream>
#include <vector>
```

```
using namespace std;
 int findPeakElement(vector<int>& nums) {
        int low=0,high=nums.size()-1;
        while(low<high){
            int mid=low+(high-low)/2;
            if(nums[mid]>nums[mid+1])
            high=mid;
            else
            low=mid+1;
        }
        return low;
    }
int main() {
int n;
cin >> n;
vector<int> nums(n);
for (int i = 0; i < n; i++) {
cin >> nums[i];
}
int result = findPeakElement(nums);
cout << result << endl;
return 0;
}
```

Expected Results:

Input: nums = [1,2,1,3,6,7,4]

Output: 5

Program4:

Problem Statement:

[Single Element in a Sorted Array](#) (Leetcode 540)

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return *the single element that appears only once*.

Your solution must run in O(log n) time and O(1) space.

Constraints:

- 1 <= nums.length <= 10^5
- 0 <= nums[i] <= 10^5

Algorithm:

Step1. Initialize low = 0 and high = n-1.

Step2. Use binary search.

- Always compare nums[mid] with its neighbor.
- Adjust mid to even index to ensure pair checking.
- If nums[mid] == nums[mid+1], the single element is after mid+1, else it's before or at mid.

Step3. Return the value at low.

Implementation:

```
#include <iostream>
#include <vector>
using namespace std;
int singleNonDuplicate(vector<int>& nums) {
       int low=0,high=nums.size()-1;
        while(low<high){
            int mid=low+(high-low)/2;
            if(mid%2==1)
            mid--;
            if(nums[mid]==nums[mid+1])
            low=mid+2;
            else
           high=mid;
        }
        return nums[low];
    }
int main() {
int n;
cin >> n;
vector<int> nums(n);
for (int i = 0; i < n; i++) {
cin >> nums[i];
}
int result = singleNonDuplicate(nums);
cout << result << endl;
return 0;
}
```

Expected Results:

Input: nums = [1,1,2,3,3,4,4,8,8]

Output: 2

Input: nums = [3,3,7,7,10,11,11]

Output: 10

Program4:

Problem Statement:

74. Search a 2D Matrix(Leetcode: 74)

You are given an m x n integer matrix with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m * n)) time complexity.

Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 100
- -10^4 <= matrix[i][j], target <= 10^4

Algorithm:

Step 1: Traverse Rows:

- Start from the first row and check if the last element of each row is greater than or equal to target.
- If the last element of the current row (matrix[i][n-1]) is greater than or equal to the target, you can continue searching in that row. Otherwise, move to the next row.
- Stop when you reach the row whose last element is greater than or equal to target or when all rows have been checked.

Step 2: Binary Search Within Row:

- For the row where the target might be, perform binary search to find the target.
    - Set the low pointer to the first column (low = 0).
    - Set the high pointer to the last column (high = n-1).
    - While low <= high, calculate the mid-point of the current row:
        - mid = low + (high - low) / 2.
    - Compare the matrix[i][mid] with target:
        - If matrix[i][mid] == target, return true.
        - If matrix[i][mid] < target, move the low pointer to mid + 1.
        - If matrix[i][mid] > target, move the high pointer to mid - 1.

Step 3: Return Result:

- If the target is found in the binary search, return true.
- If after checking all rows and performing binary searches no match is found, return false.

Implementation:

```
#include <iostream>
#include <vector>
using namespace std;
bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m=matrix.size();
        int vector<vector<int>>n=matrix[0].size();
        int i=0;
        while(i<m and matrix[i][n-1]<=target){
            if(matrix[i][n-1]==target)
                return true;
            i++;
        }
        if(i<=m){
            if(i==m){
                i=m-1;
            }
            int low=0;
            int high=n-1;
            while(low<=high){
                int mid=low+(high-low)/2;
                if(matrix[i][mid]==target)
                return true;
                else if(matrix[i][mid]<target){
                    low=mid+1;
                }
                else{
                   high=mid-1;
                }
            }
        }
        return false;
    }
int main() {
int n,m,target;
cin >> m>>n;
vector<vector<int>> matrix(m,vector<int>(n));
for (int i = 0; i < m; i++) {
for (int j = 0; i < n; j++)
cin >> nums[i][j];
}
cin>>target;
bool result= searchMatrix(matrix,target);
if(result)
```

```
cout <<"true";
else{
    cout<<"false";
}
return 0;
}
```

Expected Results:

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

Output: true

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

Output: false

Program5:

Problem Statement:

Capacity To Ship Packages Within D Days(Leetcode 1011)

A conveyor belt has packages that must be shipped from one port to another within days days.

The i^{th} package on the conveyor belt has a weight of weights[i]. Each day, we load the ship with packages on the conveyor belt (in the order given by weights). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within days days.

Constraints:

- 1 <= days <= weights.length <= $5 * 10^4$
- 1 <= weights[i] <= 500

Algorithm:

Step1: Binary Search on the Ship Capacity (maxload):

- The core of the solution is using binary search to minimize the maxload (ship capacity).
- The lower bound (low) is the maximum weight of an individual item because the ship has to carry at least that much.
- The upper bound (high) is the total sum of all the weights, because the worst-case scenario is that the ship needs to carry all items in one day.

Step2: Helper Function loadcheck:

- This function checks if it's possible to ship the weights in days days with a given maxload.
- It simulates the process of adding weights to a ship each day. If adding a weight exceeds the maxload, it starts a new day (increments dc).
- If the number of days required (dc) is less than or equal to days, the function returns true. Otherwise, it returns false.

Step3: Binary Search Loop:

- While the low is less than high, the algorithm checks the midpoint (mid) of the current range. If it is possible to ship the weights within days with the mid ship capacity, it narrows the search range to the left (lower values of maxload).
- Otherwise, it narrows the search range to the right (higher values of maxload).

Step4: Return the Optimal Ship Capacity:

- The process continues until low equals high, which will be the minimum ship capacity required.

Implementation:

```
#include <iostream>
#include <vector>
using namespace std;
bool loadcheck(vector<int>& weights, int days,int maxload)
{
    int dc=1,load=0;
    for(int w:weights){
        if(w+load>maxload){
            dc++;
            load=0;
        }
        load+=w;
    }
    return dc<=days;
}
int shipWithinDays(vector<int>& weights, int days) {
        int low=0,high=0;
        for(int w:weights){
            low=max(low,w);
            high+=w;
        }
        while(low<high){
            int mid=low+(high-low)/2;
            if(loadcheck(weights,days,mid))
            {
                high=mid;
            }
```

```
            else
            {
                low=mid+1;
            }
        }
        return low;
    }
int main() {
int n,days;
cin >>n;
vector<int> weights(n);
for (int i = 0; i < n; i++) {
cin >> weights[i];
}
cin>>days;
cout<<shipWithinDays( weights, days);
return 0;
}
```

Expected Results:

Sample 1:

Input: weights = [1,2,3,4,5,6,7,8,9,10], days = 5

Output: 15

Explanation: A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:

1st day: 1, 2, 3, 4, 5

2nd day: 6, 7

3rd day: 8

4th day: 9

5th day: 10

Note that the cargo must be shipped in the order given, so using a ship of capacity 14 and splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is not allowed.

Sample 2:

Input: weights = [3,2,2,4,1,4], days = 3

Output: 6

Sample 3:

Input: weights = [1,2,3,1,1], days = 4

Output: 3

LAB MODULE - II

Sorting

Program : 1

Aim : **Minimum Moves to Equal Array Elements**

Given an integer array nums of size n, return the minimum number of moves required to make all array elements equal. In one move, you can increment n - 1 elements of the array by 1.

Example 1:

Input: nums = [1,2,3]

Output: 3

Explanation: Only three moves are needed (remember each move increments two elements):

[1,2,3] => [2,3,3] => [3,4,3] => [4,4,4]

Example 2:

Input: nums = [1,1,1]

Output: 0

Constraints:

n == nums.length

1 <= nums.length <= 10^5

-10^9 <= nums[i] <= 10^9

The answer is guaranteed to fit in a 32-bit integer.

Algorithm:

Step 1: Read an integer n (the number of elements). Read n integers into the array nums.

Find the smallest element

Step 2: Set minVal = nums[0].

For each element val in nums:

• If val < minVal, update minVal = val.

(After this loop, minVal holds the array's minimum value.)

Accumulate the required moves

Step 3: Set result = 0.

For each element val in nums:

• Add val – minVal to result.

(Each term val – minVal is the number of decrements needed to bring that element down to minVal.)

Step 4: Return / print the answer

Output the value stored in result; this is the minimum total moves.

Program:

```
#include <iostream>
#include <vector>
```

```
using namespace std;
int minMoves(const vector<int>& nums)
{
    int minVal = nums[0];
    long long result = 0;
    // 1) Find the minimum value in the array
    for (int val : nums)
        if (val < minVal)
            minVal = val;
    // 2) Sum the differences to the minimum
    for (int val : nums)
        result += val - minVal;
    return static_cast<int>(result);
}
int main()
{
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter " << n << " integers: ";
    for (int i = 0; i < n; ++i)
    {
        cin >> nums[i];
    }
    cout << "Minimum moves: " << minMoves(nums) << '\n';
    return 0;
}
```

Input:-

Enter number of elements: 3

Enter 3 integers: 1 2 3

Output:-

Minimum moves: 3

Program : 2

Aim : **Minimum Absolute Difference**

Given an array of distinct integers arr, find all pairs of elements with the minimum absolute difference of any two elements. Return a list of pairs in ascending order(with respect to pairs), each pair [a, b] follows
1. a, b are from arr,

2. a < b

3. b - a equals to the minimum absolute difference of any two elements in arr.

Example 1:

Input: arr = [4,2,1,3]

Output: [[1,2],[2,3],[3,4]]

Explanation: The minimum absolute difference is 1. List all pairs with difference equal to 1 in ascending order.

Example 2:

Input: arr = [1,3,6,10,15]

Output: [[1,3]]

Example 3:

Input: arr = [3,8,-10,23,19,-4,-14,27]

Output: [[-14,-10],[19,23],[23,27]]

Constraints:

2 <= arr.length <= 10^5

-10^6 <= arr[i] <= 10^6

Algorithm:

Step 1: Input the array size (n): Prompt the user with “Enter number of elements:” Read an integer n.

Step 2: Input the array elements: Create a vector arr of length n. Prompt “Enter the elements:” and read n integers into arr.

Step 3: Sort the array : Rearrange arr in non-decreasing order using std::sort. After this step, any two elements that were originally neighbors in the array may now be far apart, but adjacent elements in sorted order give the smallest possible absolute differences.

Step 4: Find the global minimum absolute difference: Initialize min_diff to INT_MAX. Scan the sorted array once: for each index i from 0 to n − 2, compute diff = arr[i + 1] − arr[i].
Keep the smallest value seen in min_diff.

Step 5: Collect all pairs that achieve min_diff : Create an empty 2-D vector ans. Scan the array a second time: for each index i,

If arr[i + 1] − arr[i] == min_diff, push the pair
{arr[i], arr[i + 1]} into ans.

When this loop finishes, ans holds every adjacent pair whose difference equals the global minimum.

Step 6: Output the result: Print “Pairs with minimum absolute difference:”. For each pair in ans, print it in the format [x, y].

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;
// Function to find all pairs with the minimum absolute difference
vector<vector<int>> minimumAbsDifference(vector<int>& arr)
{
    sort(arr.begin(), arr.end());
    int min_diff = INT_MAX;
    for (size_t i = 0; i + 1 < arr.size(); ++i)
    {
        min_diff = min(min_diff, arr[i + 1] - arr[i]);
    }
    vector<vector<int>> ans;
    for (size_t i = 0; i + 1 < arr.size(); ++i)
    {
        if (arr[i + 1] - arr[i] == min_diff)
        {
            ans.push_back({arr[i], arr[i + 1]});
        }
    }
    return ans;
}
int main()
{
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i)
    {
        cin >> arr[i];
    }
    vector<vector<int>> result = minimumAbsDifference(arr);
    cout << "Pairs with minimum absolute difference:\n";
    for (const auto& pair : result)
    {
```

```
        cout << "[" << pair[0] << ", " << pair[1] << "]\n";
    }
    return 0;
}
```

Input:-

Enter number of elements: 4

Enter the elements: 4 2 1 3

Output:-

Pairs with minimum absolute difference:

[1, 2]

[2, 3]

[3, 4]

Program : 3

Aim : Sort Array By Parity

Given an integer array nums, move all the even integers at the beginning of the array followed by all the odd integers. Return any array that satisfies this condition.

Example 1:

Input: nums = [3,1,2,4]

Output: [2,4,3,1]

Explanation: The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3] would also be accepted.

Example 2:

Input: nums = [0]

Output: [0]

Constraints:

1 <= nums.length <= 5000

0 <= nums[i] <= 5000

Algorithm:

Step 1: Read array size: Store the integer in n.

Step 2: Read array elements: Create a vector nums of length n. And read n integers into nums.

Step 3: Initialize the even pointer

Set j ← 0.

Invariant: All indices < j already hold even numbers.

Step 4: Single linear scan (for i = 0 … n−1)

If nums[i] is even (nums[i] % 2 == 0):

Swap nums[i] with nums[j].

Increment j (j ← j + 1).

Else (odd), do nothing.

Effect: Every time an even value is found, it is moved (possibly with itself) to the current "front" position, and j advances.

Step 5: Return / keep the reordered array: After the loop, all evens occupy indices 0 … j−1; all odds occupy j … n−1. The function returns nums, but the rearrangement is already in-place.

Step 6: Output the result: Print "Array after sorting by parity:" and the contents of nums.

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
vector<int> sortArrayByParity(vector<int>& nums)
{
   int j = 0;                // position for next even number
   for (int i = 0; i < nums.size(); ++i)
   {
      if (nums[i] % 2 == 0)        // even?
      {
         swap(nums[i], nums[j]);  // move it forward
         ++j;
      }
   }
   return nums;  // in-place, but return for convenience
}
int main()
{
   int n;
   cout << "Enter number of elements: ";
   cin >> n;
   vector<int> nums(n);
   cout << "Enter the elements: ";
   for (int i = 0; i < n; ++i)
   {
      cin >> nums[i];
   }
   sortArrayByParity(nums);
   cout << "Array after sorting by parity:\n";
   for (int x : nums) cout << x << ' ';
   cout << '\n';
```

```
    return 0;
}
```

Input:-

Enter number of elements: 4

Enter the elements: 3 1 2 4

Output:-

Array after sorting by parity:

2 4 3 1

Program : 4

Aim : Max Chunks To Make Sorted

You are given an integer array arr of length n that represents a permutation of the integers in the range [0, n - 1]. We split arr into some number of chunks (i.e., partitions), and individually sort each chunk. After concatenating them, the result should equal the sorted array. Return the largest number of chunks we can make to sort the array.

Example 1:

Input: arr = [4,3,2,1,0]

Output: 1

Explanation: Splitting into two or more chunks will not return the required result. For example, splitting into [4, 3], [2, 1, 0] will result in [3, 4, 0, 1, 2], which isn't sorted.

Example 2:

Input: arr = [1,0,2,3,4]

Output: 4

Explanation: We can split into two chunks, such as [1, 0], [2, 3, 4]. However, splitting into [1, 0], [2], [3], [4] is the highest number of chunks possible.

Constraints:

n == arr.length

1 <= n <= 10

0 <= arr[i] < n

All the elements of arr are unique.

Algorithm:

Step 1: Read the array size: Store the user's input in the integer n.

Step 2: Read the array elements: Create a vector arr with length n. Read n integers into arr.

Step 3: Initialize counters:

Set chunks ← 0 — counts how many valid chunks are found.

Set maxSoFar ← 0 — keeps the maximum value seen while scanning.

Step 4: Single left-to-right scan of the array

For each index i from 0 to n − 1:

Update running maximum

maxSoFar ← max(maxSoFar, arr[i]).

Check chunk cut condition

If maxSoFar == i, then every value observed up to position i is ≤ i, which means the subarray [0 … i] already contains exactly the set {0, 1, …, i}.

Action: increment chunks.

Step 5: Return or print the result: After the loop, chunks equals the maximum number of segments such that sorting each segment individually and concatenating them gives a fully sorted array.

Print “Maximum chunks: X” where X is the final count.

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int maxChunksToSorted(const vector<int>& arr)
{
   int chunks   = 0;
   int maxSoFar = 0;
   for (int i = 0; i < static_cast<int>(arr.size()); ++i)
   {
      maxSoFar = max(maxSoFar, arr[i]);
      if (maxSoFar == i)            // all elements up to i are ≤ i
      {
         ++chunks;                  // we can cut a chunk here
      }
   }
   return chunks;
}
int main()
{
   int n;
   cout << "Enter number of elements: ";
   cin >> n;
   vector<int> arr(n);
   cout << "Enter the elements (0 to " << n - 1 << " in any order): ";
   for (int i = 0; i < n; ++i)
   {
      cin >> arr[i];
```

```
    }
    int result = maxChunksToSorted(arr);
    cout << "Maximum chunks: " << result << '\n';
    return 0;
}
```

Input:-

Enter number of elements: 5

Enter the elements (0 to 4 in any order): 4 3 2 1 0

Output:-

Maximum chunks: 1

Program : 5

Aim : Kth Largest Element in an Array

Given an integer array nums and an integer k, return the kth largest element in the array. Note that it is the kth largest element in the sorted order, not the kth distinct element. Can you solve it without sorting?

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2

Output: 5

Example 2:

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4

Output: 4

Constraints:

1 <= k <= nums.length <= 10^5

-10^4 <= nums[i] <= 10^4

Algorithm:

Step 1: Input the array size: Store the value in n.

Step 2: Input the array elements: Create a vector nums with length n. Read the n integers into nums.

Step 3: Input k: Store the value in k. Goal: find the k-th largest element of nums.

Step 4: Prepare the frequency table: Define two constants

SHIFT ← 10000 (to make negative numbers non-negative)

SIZE ← 20001 (covers the shifted range 0 … 20000).

Declare an integer array freq[SIZE], initialized to 0.

Step 5: Populate the frequency table: For each element x in nums:

Compute index ← x + SHIFT.

Increment freq[index].

Step 6: Scan the frequency table from high to low:

For i from SIZE − 1 down to 0:

Check if current bucket contains the answer

If freq[i] ≥ k, then the k-th largest value is i − SHIFT. Return that value.

Otherwise subtract the entire bucket and continue:

k ← k − freq[i] (we have skipped all occurrences of value i − SHIFT).

Step 7: Output the result: Print "The k-th largest element is: ans", where ans is the value returned in step 6.

Program:

```
#include <iostream>
#include <vector>
using namespace std;
int findKthLargest(const vector<int>& nums, int k)
{
   const int SHIFT = 10000;      // shift to make indices non-negative
   const int SIZE  = 20001;      // covers range [-10000, 10000]
   int freq[SIZE] = {0};         // frequency table

   for (int x : nums)            // Build frequency table
   {
      ++freq[x + SHIFT];
   }
  for (int i = SIZE - 1; i >= 0; --i)  // Scan from high to low to find the k-th largest
  {
      if (freq[i] >= k)
      {
         return i - SHIFT;     // convert index back to original value
      }
      k -= freq[i];
   }
   return 0;
}
int main()
{
   int n, k;
   cout << "Enter number of elements: ";
   cin >> n;
   vector<int> nums(n);
   cout << "Enter " << n << " integers (each between -10000 and 10000): ";
   for (int i = 0; i < n; ++i)
   {
```

```
        cin >> nums[i];
    }
    cout << "Enter k (1 ≤ k ≤ n): ";
    cin >> k;
    int ans = findKthLargest(nums, k);
    cout << "The " << k << "-th largest element is: " << ans << '\n';
    return 0;
}
```

Input:

Enter number of elements: 6

Enter 6 integers (each between -10000 and 10000): 3 2 1 5 6 4

Enter k (1 ≤ k ≤ n): 2

Output:

The 2-th largest element is: 5

Program : 6

Aim : Boats to Save People

You are given an array people where people[i] is the weight of the ith person, and an infinite number of boats where each boat can carry a maximum weight of limit. Each boat carries at most two people at the same time, provided the sum of the weight of those people is at most limit. Return the minimum number of boats to carry every given person.

Example 1:

Input: people = [1,2], limit = 3

Output: 1

Explanation: 1 boat (1, 2)

Example 2:

Input: people = [3,2,2,1], limit = 3

Output: 3

Explanation: 3 boats (1, 2), (2) and (3)

Example 3:

Input: people = [3,5,3,4], limit = 5

Output: 4

Explanation: 4 boats (3), (3), (4), (5)

Constraints:

1 <= people.length <= 5 * 10^4

1 <= people[i] <= limit <= 3 * 10^4

Algorithm:

Step 1: Read input data: Store the value in n. Create a vector people of length n.

Prompt “Enter the weights of n people:” and read the n weights into people.

Prompt “Enter boat weight limit:” and read the integer limit.

Step 2: Sort the weights: Sort people in ascending order. After sorting, the lightest person is at index 0 and the heaviest at index n − 1.

Step 3: Initialize two-pointer indices and counter:

i ← 0 // points to the lightest remaining person

j ← n − 1 // points to the heaviest remaining person

boats ← 0 // counts boats used so far

Step 4: Greedy pairing loop

While i ≤ j (people still unassigned):

Attempt to pair: If people[i] + people[j] ≤ limit, the lightest (i) can share a boat with the heaviest (j). Increment i to skip the lightest (they board).

Assign the heaviest person: Decrement j (the heaviest boards in any case, alone or with partner).

Count the boat: Increment boats because a new boat has been used.

Step 5: Output the result:

Print “Minimum boats needed:” followed by the value in boats.

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int numRescueBoats(vector<int>& people, int limit)
{
   sort(people.begin(), people.end());          // ascending order
   int i = 0;                                   // lightest person
   int j = static_cast<int>(people.size()) - 1;  // heaviest
   int boats = 0;
   while (i <= j)
   {
      if (people[i] + people[j] <= limit)  // can pair lightest + heaviest
         ++i;                              // lightest person boards too
      --j;                                 // heaviest person boards
      ++boats;                             // one boat used
   }
   return boats;
}
int main()
```

```
{
    int n, limit;
    cout << "Enter number of people: ";
    cin >> n;
    vector<int> people(n);
    cout << "Enter the weights of " << n << " people: ";
    for (int i = 0; i < n; ++i)
    {
      cin >> people[i];
    }
    cout << "Enter boat weight limit: ";
    cin >> limit;
    int result = numRescueBoats(people, limit);
    cout << "Minimum boats needed: " << result << '\n';
    return 0;
}
```

Input:-

Enter number of people: 4

Enter the weights of 4 people: 3 2 2 1

Enter boat weight limit: 3

Output:

Minimum boats needed: 3

Lab Module – III

Recursion and Backtracking

1. Subsets: https://leetcode.com/problems/subsets/description/

Aim: Given an integer array nums of unique elements, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

Example 2:

Input: nums = [0]

Output: [[],[0]]

Constraints:

1 <= nums.length <= 10

-10 <= nums[i] <= 10

All the numbers of nums are unique.

Algorithm:

1. Input:
    - Read an integer n (number of elements).
    - Read a vector nums of size n.
2. Initialize:
    - total = 1 << n → This is 2^n, the total number of subsets.
    - Create an empty result vector to store all subsets.
3. Generate subsets using bitmasking:
    - For each mask from 0 to 2^n - 1:
        - Initialize an empty subset.
        - For each bit position i (0 to n-1):
            - If the ith bit in mask is set (i.e., mask & (1 << i)), include nums[i] in subset.
4. Add each subset to the result.
5. Print the subsets.

Program:

```
#include <iostream>
    #include <vector>
    using namespace std;

    void sub(const vector<int>& nums, int start, vector<int>& current,
vector<vector<int>>& result) {
        result.push_back(current);
        for (int i = start; i < nums.size(); ++i) {
            current.push_back(nums[i]);
            sub(nums, i + 1, current, result);
            current.pop_back();
        }
    }

    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> current;
        sub(nums, 0, current, result);
        return result;
    }

    int main() {
        int n;
        cin >> n;

        vector<int> nums(n);
        for (int i = 0; i < n; i++) {
            cin >> nums[i];
        }

        vector<vector<int>> all_subsets = subsets(nums);
        for (const auto& subset : all_subsets) {
            cout << "{ ";
            for (int num : subset) {
                cout << num << " ";
            }
            cout << "}\n";
        }

        return 0;
    }
```

Output:

```
3
1 2 3
{ }
{ 1 }
{ 2 }
{ 1 2 }
{ 3 }
{ 1 3 }
{ 2 3 }
{ 1 2 3 }
```

2. Permutations: https://leetcode.com/problems/permutations/description/

Aim : Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Example 2:

Input: nums = [0,1]

Output: [[0,1],[1,0]]

Example 3:

Input: nums = [1]

Output: [[1]]

Constraints:

1 <= nums.length <= 6

-10 <= nums[i] <= 10

All the integers of nums are unique.

Algorithm:

1. Input:
    - Read an integer n (number of elements).
    - Read a vector nums of size n.
2. Recursion:
    - Start from position pos = 0.
    - If pos == n - 1, add the current permutation to the result.
    - Else, for each index j from pos to n-1:
        - Swap nums[pos] with nums[j].
        - Recursively call permute with pos + 1.
        - Swap back to backtrack.
3. Return all generated permutations.

Program:

```
#include <iostream>
#include <vector>
using namespace std;

void permute(vector<int>& nums, int pos, int n, vector<vector<int>>& res)
{
    if (pos == n - 1) {
        res.push_back(nums);
        return;
    }
    for (int j = pos; j < n; j++) {
        swap(nums[pos], nums[j]);
        permute(nums, pos + 1, n, res);
        swap(nums[pos], nums[j]);
    }
}

vector<vector<int>> permute(vector<int>& nums) {
    int n = nums.size();
    vector<vector<int>> res;
    permute(nums, 0, n, res);
    return res;
}

int main() {
    int n;
    cin >> n;

    vector<int> nums(n);
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    vector<vector<int>> result = permute(nums);
    for (const auto& perm : result) {
        for (int num : perm) {
            cout << num << ' ';
        }
        cout << '\n';
    }

    return 0;
}
```

Output:

```
3
1 2 3

1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
```

3. Permutations II: https://leetcode.com/problems/permutations-ii/description/

Aim : Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.

Example 1:

Input: nums = [1,1,2]

Output:

[[1,1,2],

[1,2,1],

[2,1,1]]

Example 2:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Constraints:

1 <= nums.length <= 8

-10 <= nums[i] <= 10

Algorithm:

1. Input:
    - Read integer n (number of elements).
    - Read vector nums of size n.
2. Backtracking with swapping:
    - Recursive function permute(nums, pos, n, res):
        - If pos == n-1, insert current nums permutation into a set.
        - Else, for each index i from pos to n-1:
            - Swap nums[pos] with nums[i].
            - Recursively call permute with pos+1.
            - Swap back to restore original order (backtrack).
3. Avoid duplicates by using a set:
    - The set<vector<int>> automatically stores unique permutations only.
4. Convert the set to vector for returning final results.

Program:

```
#include <iostream>
    #include <vector>
    #include <set>
    #include <algorithm>
    using namespace std;
    void permute(vector<int>& nums, int pos, int n, set<vector<int>>& res) {
        if (pos == n - 1) {
            res.insert(nums);
            return;
        }
        for (int i = pos; i < n; ++i) {
            swap(nums[pos], nums[i]);
            permute(nums, pos + 1, n, res);
            swap(nums[pos], nums[i]);
        }
    }
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> ans;
        set<vector<int>> res;
        permute(nums, 0, n, res);
        ans.assign(res.begin(), res.end());
        return ans;
    }

    int main() {
        int n;
        cin >> n;

        vector<int> nums(n);
        for (int i = 0; i < n; ++i) {
            cin >> nums[i];
        }

        vector<vector<int>> result = permuteUnique(nums);
        for (const auto& vec : result) {
            for (int num : vec) {
                cout << num << " ";
            }
            cout << "\n";
        }

        return 0;
    }
```

Output:

```
3
1 1 2
1 1 2
1 2 1
2 1 1
```

4. Subsets II: https://leetcode.com/problems/subsets-ii/description/

Aim: Given an integer array nums that may contain duplicates, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,2]

Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]

Example 2:

Input: nums = [0]

Output: [[],[0]]

Constraints:

1 <= nums.length <= 10

-10 <= nums[i] <= 10

Algorithm:

1. Sort the input array to group duplicates together.
2. Define a recursive function:
    - Parameters: current position pos, current subset ans, and result set res.
    - Base case: If pos == nums.size(), insert the current subset ans into res.
    - Recursive case:
        - Include nums[pos] in ans and recurse to next position.
        - Exclude nums[pos] from ans and recurse to next position.
3. Store subsets in a set to remove duplicates automatically.
4. After recursion completes, convert the set into a vector and return it.

Program:

```
#include <iostream>
    #include <vector>
    #include <set>
    #include <algorithm>
    using namespace std;

    void create(vector<int>& nums, int pos, set<vector<int>>& res,
vector<int>& ans) {
        if (pos == nums.size()) {
            res.insert(ans);
            return;
        }
        ans.push_back(nums[pos]);
        create(nums, pos + 1, res, ans);
        ans.pop_back();
        create(nums, pos + 1, res, ans);
    }

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        set<vector<int>> res;
        vector<int> ans;
        sort(nums.begin(), nums.end());
        create(nums, 0, res, ans);
        return vector<vector<int>>(res.begin(), res.end());
    }

    int main() {
        int n;
        cin >> n;

        vector<int> nums(n);
        for (int i = 0; i < n; ++i) {
            cin >> nums[i];
        }

        vector<vector<int>> subsets = subsetsWithDup(nums);
        for (const auto& subset : subsets) {
            cout << "{ ";
            for (int num : subset) {
                cout << num << " ";
            }
            cout << "}\n";
        }

        return 0;
    }
```

Output:

```
3
1 2 2
{ }
{ 1 }
{ 1 2 }
{ 1 2 2 }
{ 2 }
{ 2 2 }
```

5. Palindrome Partitioning:
https://leetcode.com/problems/palindrome/partitioning/description/

Aim: Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s.

Example 1:

Input: s = "aab"

Output: [["a","a","b"],["aa","b"]]

Example 2:

Input: s = "a"

Output: [["a"]]

Constraints:

1 <= s.length <= 16

s contains only lowercase English letters.

Algorithm:

1. Define a helper function isPalindrome(s, start, end) that checks if the substring s[start..end] is a palindrome.
2. Use a recursive function partition(index, s, path, ans):
    - index = current starting position in the string
    - path = current list of palindromic substrings
    - ans = final list of palindrome partitions
3. Base case:
    - If index == length of s, push path into ans.
4. Recursive case:
    - For every end position i from index to s.size() - 1:
        - Check if s[index..i] is palindrome.
        - If yes, append s[index..i] to path, recurse with i + 1.
        - Backtrack by removing the last substring added to path.

Program:

```
#include <bits/stdc++.h>
using namespace std;

bool isPalindrome(string s, int start, int end) {
    while (start <= end) {
      if (s[start++] != s[end--])
        return false;
    }
    return true;
}
void partition(int index, string s, vector < string > & path,
    vector < vector < string > > & ans) {

    if (index == s.size()) {
      ans.push_back(path);
      return;
    }
    for (int i = index; i < s.size(); ++i) {
      if (isPalindrome(s, index, i)) {
        path.push_back(s.substr(index, i - index + 1));
        partition(i + 1, s, path, ans);
        path.pop_back();
      }
    }
  }


int main() {
  string s;
  cin>>s;
  vector < vector < string >> ans;
  vector < string > path;
  partition(0,s,path,ans);
  int n = ans.size();
  cout << " [ ";
  for (auto i: ans) {
    cout << "[ ";
    for (auto j: i) {
      cout <<"'"<< j <<"'"<< " ";
    }
    cout << "] ";
  }
  cout << "]";

  return 0;
}
```

Output:

```
aab
[ [ 'a' 'a' 'b' ] [ 'aa' 'b' ] ]
```

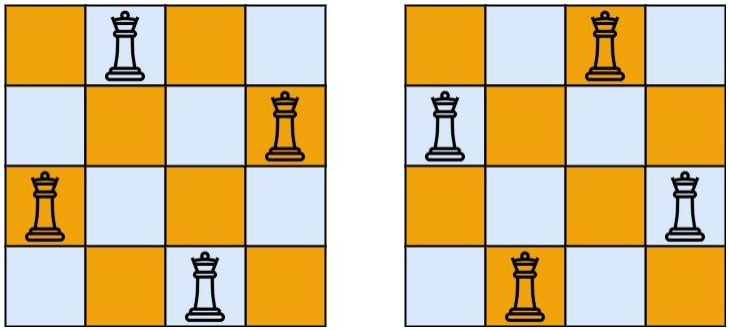
6. N-Queens: https://leetcode.com/problems/n-queens/description/

Aim : The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1:



Input: n = 4

Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]

Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above

Example 2:

Input: n = 1

Output: [["Q"]]

Constraints:

1 <= n <= 9

Algorithm:

1. Initialize an empty n x n board with zeros (0 means no queen, 1 means queen).
2. Define issafe(board, x, y, n) to check if placing a queen at (x, y) is safe:
    - Check the column y upwards.
    - Check the upper-left diagonal.
    - Check the upper-right diagonal.
3. Backtracking function queen(board, row, n, res):

    - If row == n, all queens are placed, add board to results.
    - Otherwise, for each column in the current row:
        - Check if safe to place queen at (row, col).
        - If yes, place queen and recurse for next row.
        - Remove queen to backtrack.
4. Start from row 0 and call the backtracking function.
5. Print all valid solutions at the end.

Program:

```
#include <iostream>
    #include <vector>
    using namespace std;

    bool issafe(vector<vector<int>> &board, int x, int y, int n) {
        for (int row = 0; row < x; row++) {
            if (board[row][y] == 1)
                return false;
        }
        int row = x, col = y;
        while (row >= 0 && col >= 0) {
            if (board[row][col] == 1)
                return false;
            row--;
            col--;
        }
               row = x, col = y;
        while (row >= 0 && col < n) {
            if (board[row][col] == 1)
                return false;
            row--;
            col++;
        }
        return true;
    }

    void queen(vector<vector<int>> &board, int row, int n,
vector<vector<vector<int>>> &res) {
        if (row == n) {
            res.push_back(board);
            return;
        }
        for (int col = 0; col < n; col++) {
            if (issafe(board, row, col, n)) {
                board[row][col] = 1;
                queen(board, row + 1, n, res);
                board[row][col] = 0;
            }
        }
    }

    int main() {
        int n;
        cin >> n;
        vector<vector<int>> board(n, vector<int>(n, 0));
        vector<vector<vector<int>>> res;
```

```
        queen(board, 0, n, res);

        for (int k = 0; k < res.size(); k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    cout << (res[k][i][j] ? "Q " : ". ");
                }
                cout << endl;
            }
            cout << endl;
        }

        return 0;
    }
```

Output :

```
4
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

LAB MODULE - IV

Linked List Part 1

Program : 1

Aim : **Middle of the Linked List**

Given the head of a singly linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.

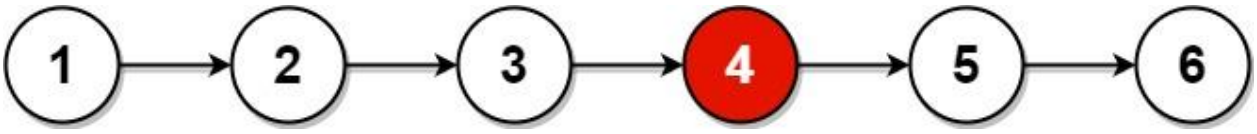
Example 1:

Input: head = [1,2,3,4,5]

Output: [3,4,5]

Explanation: The middle node of the list is node 3.

Example 2:



Input: head = [1,2,3,4,5,6]

Output: [4,5,6]

Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.

Constraints:

The number of nodes in the list is in the range [1, 100].

1 <= Node.val <= 100

Algorithm:

Step 1: Create a ListNode structure with two members:

val: An integer to hold the node's value.

next: A pointer to the next node in the list.

Step 2: Build the Linked List In the main( ) function:

Dynamically create five nodes with values: 1 → 2 → 3 → 4 → 5.

Link them to form a singly linked list using the next pointer.

Step 3: Initialize the Solution Class

Create an instance of the Solution class to access the middleNode function.

Step 4: Count the Length of the Linked List

Inside middleNode(ListNode* head):

Initialize a pointer current to the head of the list.

Initialize a counter variable length to 0.

Traverse the list: While current is not null, move to the next node and increment length. After traversal, length contains the total number of nodes in the list.

Step 5: Calculate the Middle Position ,compute the middle index as: middle = length / 2;

Step 6: Traverse to the Middle Node: Reset current back to the head.

Move current forward middle number of steps using a loop:

```
for (int i = 0; i < middle; i++)
    current = current->next;
```

After the loop, current points to the middle node.

Step 7: Return the Middle Node: Return the current node (middle node) from the function.

Step 8: Store the returned middle node in a pointer named middle. Print its value: middle->val

Step 9: Use a loop to delete all nodes in the linked list to prevent memory leaks.

Program:

```
#include <iostream>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        int length = 0;
        ListNode* current = head;
        while (current != nullptr) {
            current = current->next;
            length++;
        }
        int middle = length / 2;
        current = head;
        for (int i = 0; i < middle; i++) {
            current = current->next;
        }
        return current;
    } };
int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
```

```
    head->next->next->next->next = new ListNode(5);
    Solution sol;
    ListNode* middle = sol.middleNode(head);
    cout << "The middle node value is: " << middle->val << endl;
    while (head != nullptr) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
    }
    return 0;
}
```

Output:-

The middle node value is: 3

Program : 2

Aim : **Delete Node in a Linked List**

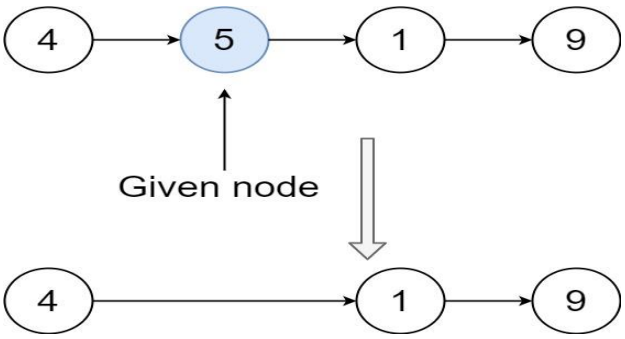
There is a singly-linked list head and we want to delete a node node in it. You are given the node to be deleted node. You will not be given access to the first node of head. All the values of the linked list are unique, and it is guaranteed that the given node node is not the last node in the linked list. Delete the given node. Note that by deleting the node, we do not mean removing it from memory. We mean:

The value of the given node should not exist in the linked list. The number of nodes in the linked list should decrease by one. All the values before node should be in the same order. All the values after node should be in the same order.

Custom testing:

For the input, you should provide the entire linked list head and the node to be given node. node should not be the last node of the list and should be an actual node in the list. We will build the linked list and pass the node to your function. The output will be the entire list after calling your function.

Example 1:

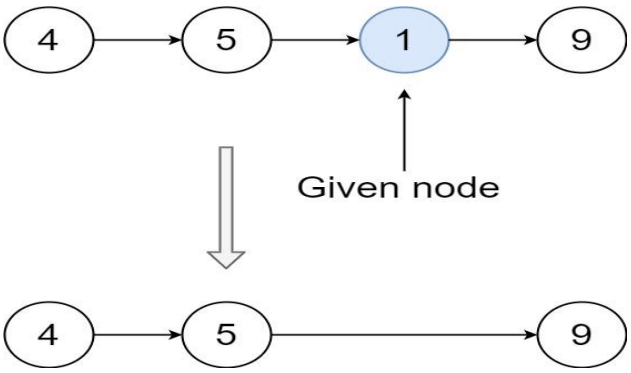


Input: head = [4,5,1,9], node = 5

Output: [4,1,9]

Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

Example 2:



Input: head = [4,5,1,9], node = 1

Output: [4,5,9]

Explanation: You are given the third node with value 1, the linked list should become 4 -> 5 -> 9 after calling your function.

Constraints:

The number of the nodes in the given list is in the range [2, 1000].

-1000 <= Node.val <= 1000

The value of each node in the list is unique.

The node to be deleted is in the list and is not a tail node.

Algorithm:

Step 1: Create a structure ListNode with: An integer val to store data. A pointer next to point to the next node.

Step 2: Define the deleteNode Function: Inside the Solution class, define deleteNode(ListNode* node): Copy the value from the next node into the current node:

node->val = node->next->val;

Bypass the next node by updating the current node's next pointer:

node->next = node->next->next;

Step 3: Define Helper Function printList

Iterate over the linked list from the given head.

Print each node's value followed by " -> " if it's not the last node.

Step 4: Define Helper Function findNode: Traverse the list starting from head. Compare each node's value with the targetValue. Return the pointer to the node when a match is found.

Step 5: Create and Initialize the Linked List in main(): Manually create nodes with values: 4 -> 5 -> 1 -> 9. Link them using next pointers.

Step 6: Display the Original List: Call printList(head) to print the initial list.

Step 7: Find the Node to Delete: Use findNode(head, targetValue) with targetValue = 5 to locate the node.

Step 8: Check If Deletion Is Valid: The node to delete is not null. The node is not the last node (i.e., node->next != nullptr).

Step 9: Delete the Node: Call deleteNode(nodeToDelete) using a Solution object.

Step 10: Display the Updated List: Call printList(head) again to show the result after deletion.

Step 11: Free Allocated Memory: Traverse the list and delete each node to avoid memory leaks.

Program:

```
#include <iostream>
using namespace std;
struct ListNode
{
   int val;
   ListNode* next;
   ListNode(int x) : val(x), next(nullptr) {}
};
class Solution
{
public:
   void deleteNode(ListNode* node) {
      node->val = node->next->val;
      node->next = node->next->next;
   }
};
void printList(ListNode* head) {
   ListNode* current = head;
   while (current != nullptr) {
      cout << current->val;
      if (current->next != nullptr) cout << " -> ";
      current = current->next;
   }
   cout << endl;
}
ListNode* findNode(ListNode* head, int value) {
   ListNode* current = head;
   while (current != nullptr && current->val != value) {
      current = current->next;
   }
```

```
    return current;
}
int main() {
    ListNode* head = new ListNode(4);
    head->next = new ListNode(5);
    head->next->next = new ListNode(1);
    head->next->next->next = new ListNode(9);
    cout << "Original list: ";
    printList(head);
    int targetValue = 5;
    ListNode* nodeToDelete = findNode(head, targetValue);
    if (nodeToDelete != nullptr && nodeToDelete->next != nullptr) {
        Solution sol;
        sol.deleteNode(nodeToDelete);
        cout << "List after deleting node with value " << targetValue << ": ";
        printList(head);
    } else {
        cout << "Node to delete is invalid or is the last node." << endl;
    }
    while (head != nullptr) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
    }
    return 0;
}
```

Output:-

Original list: 4 -> 5 -> 1 -> 9

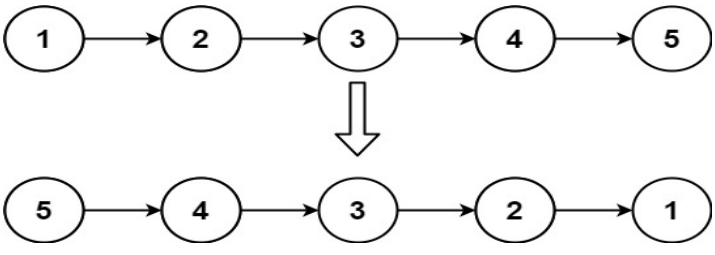
List after deleting node with value 5: 4 -> 1 -> 9

Program : 3

Aim : Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

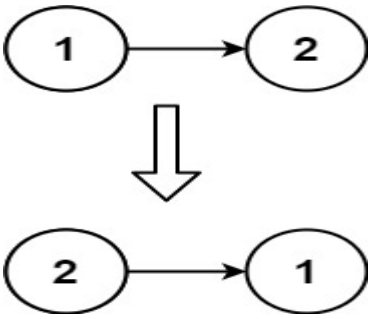
Example 1:



Input: head = [1,2,3,4,5]

Output: [5,4,3,2,1]

Example 2:



Input: head = [1,2]

Output: [2,1]

Constraints:

The number of nodes in the list is the range [0, 5000]. -5000 <= Node.val <= 5000

Algorithm:

Step 1: Define the Node Structure: Create a ListNode structure to represent a node in the singly linked list. Each node contains: An integer value val. A pointer next to the next node.

Step 2: Create the Linked List (createList)

Input: A vector of integers. If the input vector is empty, return NULL. Create the head node using the first element. Iterate through the rest of the vector: Create a new node for each element. Link the new node to the previous node. Return the head of the constructed linked list.

Step 3: Print the Linked List (printList): Traverse the list from the head node.

Print each node's value.

Step 4: Reverse the List Values using Stack (reverseList)

Input: Head of the singly linked list.

Initialize a temporary pointer temp to the head.

Create an empty stack st.

Pass 1: Push all node values onto the stack. Traverse the list: Push each node’s val onto the stack. Move to the next node.

Pass 2: Pop from stack and overwrite node values

Initialize another pointer current to the head again.

Traverse the list again:

Pop values from the stack and assign them to the current node’s val.

Move to the next node.

Return the modified head of the list.

Step 5: main() Execution

Define a vector values = {1, 2, 3, 4, 5}. Create a linked list from this vector using createList.

Print the original list. Create an object of class Solution. Call reverseList and store the result.

Print the reversed list.

Program:

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* temp = head;
        stack<int> st;
        while (temp != NULL) {
            st.push(temp->val);
            temp = temp->next;
        }
        ListNode* current = head;
        while (current != NULL && !st.empty()) {
            current->val = st.top();
            st.pop();
            current = current->next;
        }
        return head;
    }
};
ListNode* createList(const vector<int>& values) {
    if (values.empty()) return NULL;
    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;

    for (size_t i = 1; i < values.size(); ++i) {
        current->next = new ListNode(values[i]);
        current = current->next;
    }
    return head;
```

```
}
void printList(ListNode* head) {
    ListNode* current = head;
    while (current != NULL) {
        cout << current->val;
        if (current->next != NULL) cout << " -> ";
        current = current->next;
    }
    cout << endl;
}
int main() {
    vector<int> values = {1, 2, 3, 4, 5};
    ListNode* head = createList(values);
    cout << "Original list: ";
    printList(head);
    Solution solution;
    ListNode* reversedHead = solution.reverseList(head);
    cout << "Reversed list: ";
    printList(reversedHead);
    return 0;
}
```

Output:-

Original list: 1 -> 2 -> 3 -> 4 -> 5

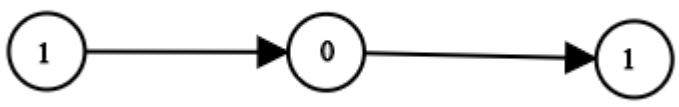
Reversed list: 5 -> 4 -> 3 -> 2 -> 1

Program : 4

Aim : **Convert Binary Number in a Linked List to Integer**

Given head which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number. Return the decimal value of the number in the linked list. The most significant bit is at the head of the linked list.

Example 1:



Input: head = [1,0,1]

Output: 5

Explanation: (101) in base 2 = (5) in base 10

Example 2:

Input: head = [0]

Output: 0

Constraints:

The Linked List is not empty.

Number of nodes will not exceed 30.

Each node's value is either 0 or 1.

Algorithm:

Step 1: Define Node Structure: Define a struct ListNode to represent a node in the singly linked list.

Each node contains: An integer value val (either 0 or 1). A pointer next to the next node.

Step 2: Create the Linked List:

Function: createList(const vector<int>& values)

Input: A vector of integers (only 0s and 1s).

If the vector is empty, return NULL.

Create a head node with the first value.

Iterate through the rest of the vector: Create new nodes. Link each new node to the previous one.

Return the head of the list.

Step 3: Print the Linked List

Function: printList(ListNode* head). Start at the head of the list. Print each node's value.

Stop when you reach the end of the list (NULL).

Step 4: Convert Binary to Decimal

Class: Solution

Function: getDecimalValue(ListNode* head)

Initialize: Stack st to hold binary digits.

Integer multiplier = 1 for place value (powers of 2).

Integer num = 0 for the result.

Traverse the linked list from head to end: Push each node's value onto the stack. Now the stack contains bits in reverse order (least significant bit on top).

While the stack is not empty: Pop the top value from the stack. Multiply it by the current multiplier. Add it to num.

Double the multiplier (move to next power of 2). Return the result num.

Step 5: Main Function Execution

Create a vector binary = {1, 0, 1} (represents binary number 101). Call createList to construct the linked list.

Print the linked list using printList.

Create a Solution object and call getDecimalValue.

Print the result (expected output is 5 because binary 101 = decimal 5).

Program:

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        stack<int> st;
        int multiplier = 1;
        int num = 0;
        while (head != NULL) {
            st.push(head->val);
            head = head->next;
        }
        while (!st.empty()) {
            num += st.top() * multiplier;
            st.pop();
            multiplier *= 2;
        }
        return num;
    }
};
ListNode* createList(const vector<int>& values) {
    if (values.empty()) return NULL;
    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;
    for (size_t i = 1; i < values.size(); ++i) {
        current->next = new ListNode(values[i]);
        current = current->next;
    }
    return head;
}
```

```
void printList(ListNode* head) {
    while (head != NULL) {
        cout << head->val;
        if (head->next != NULL) cout << " -> ";
        head = head->next;
    }
    cout << endl;
}
int main() {
    vector<int> binary = {1, 0, 1};
    ListNode* head = createList(binary);
    cout << "Binary linked list: ";
    printList(head);
    Solution solution;
    int decimalValue = solution.getDecimalValue(head);
    cout << "Decimal value: " << decimalValue << endl;
    return 0;
}
```

Output:-

Binary linked list: 1 -> 0 -> 1

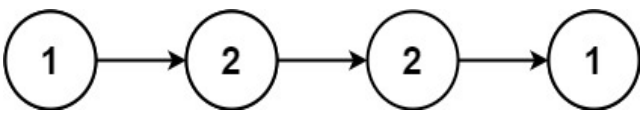
Decimal value: 5

Program : 5

Aim : **Palindrome Linked List**

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

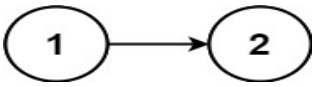
Example 1:



Input: head = [1,2,2,1]

Output: true

Example 2:



Input: head = [1,2]

Output: false

Constraints:

The number of nodes in the list is in the range [1, 10^5].

0 <= Node.val <= 9

Algorithm:

Step 1: Define Node Structure: A structure ListNode is defined to represent a node in the linked list.

Each node contains: An integer value val. A pointer next to the next node

Step 2: Create Linked List (createList)

Input: A vector of integers (e.g., {1, 2, 2, 1}).

If the vector is empty, return NULL. Create a head node with the first value.

For each subsequent value in the vector: Create a new node. Link the new node to the current node.

Move the current pointer to the new node. Return the head of the created linked list.

Step 3: Print the Linked List (printList): Start from the head node.

Traverse through the list: Print each node's val. Stop when NULL is reached.

Step 4: Check Palindrome (isPalindrome): Initialize an empty vector store to store node pointers.

Traverse the linked list: Push each node into the vector.

Initialize two pointers:

i starting from the beginning of the vector (0)

j starting from the end of the vector (size - 1)

Loop until i < j: Compare store[i]->val with store[j]->val

If they are not equal, return false (not a palindrome)

Increment i and decrement j

If all values match, return true (it is a palindrome)

Step 5: Main Function Execution (main)

Create a vector input = {1, 2, 2, 1}

Call createList(input) to build the linked list

Print the list using printList()

Create a Solution object

Call isPalindrome(head) to check if it's a palindrome

Print the result: "true" or "false"

Program:

```
#include <iostream>
#include <vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
    bool isPalindrome(ListNode* head) {
```

```
        vector<ListNode*> store;
        while (head != NULL) {
            store.push_back(head);
            head = head->next;
        }
        int j = store.size() - 1;
        for (int i = 0; i < store.size() / 2; i++) {
            if (store[i]->val != store[j]->val)
                return false;
            j--;
        }
        return true;
    }
};
ListNode* createList(const vector<int>& values) {
    if (values.empty()) return NULL;
    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;
    for (size_t i = 1; i < values.size(); ++i) {
        current->next = new ListNode(values[i]);
        current = current->next;
    }
    return head;
}
void printList(ListNode* head) {
    while (head != NULL) {
        cout << head->val;
        if (head->next != NULL) cout << " -> ";
        head = head->next;
    }
    cout << endl;
}
int main() {
    vector<int> input = {1, 2, 2, 1}; // Example input
    ListNode* head = createList(input);
    cout << "Linked list: ";
    printList(head);
    Solution solution;
```

```
    bool result = solution.isPalindrome(head);
cout << "Is palindrome? " << (result ? "true" : "false") << endl;
    return 0;
}
```

Output:

Linked list: 1 -> 2 -> 2 -> 1

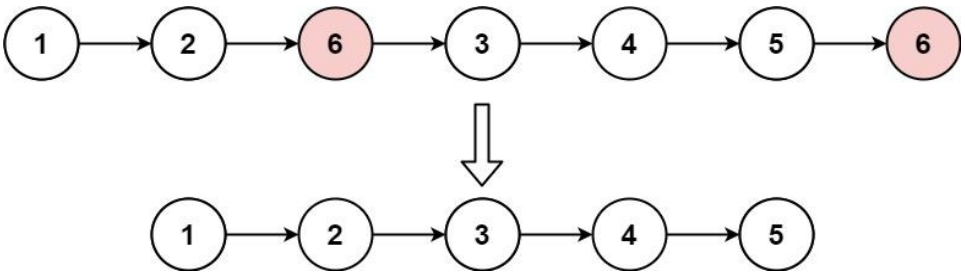
Is palindrome? true

Program : 6

Aim : **Remove Linked List Elements**

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node. val = = val, and return the new head.

Example 1:



Input: head = [1,2,6,3,4,5,6], val = 6

Output: [1,2,3,4,5]

Example 2:

Input: head = [7,7,7,7], val = 7

Output: [ ]

Constraints:

The number of nodes in the list is in the range [0, 10^4].

1 <= Node.val <= 50

0 <= val <= 50

Algorithm:

Step 1: Define the Node Structure: A struct ListNode is used to define a node in the linked list.

Each node contains: val → the integer value of the node.

next → pointer to the next node.

An additional constructor ListNode(int x, ListNode* next) is defined for easier dummy node creation.

Step 2: Create the Linked List (createList function)

Input: A vector of integers (e.g., {1, 2, 6, 3, 4, 5, 6}). If the vector is empty, return NULL. Create the head node using the first element.

Loop through the remaining elements:

Create a new node for each.

Link it to the current node.

Move the current pointer forward.

Return the head pointer.

Step 3: Print the Linked List (printList function)

If the head is NULL, print [].

Otherwise: Traverse the list.

Print each node's value.

End with a newline.

Step 4: Remove Nodes (removeElements function in Solution class)

Create a dummy node with value 0 that points to the head of the list.

Initialize:

prev → pointing to the dummy node.

ptr → pointing to the actual head of the list.

Traverse the list using ptr:

If ptr->val matches the value to be removed (val):

Skip the node by updating prev->next to ptr->next.

Else:

Move prev forward to ptr.

Move ptr forward in every iteration.

Return dummy.next, which points to the new head of the modified list.

Step 5: Main Function Logic: Define input vector: {1, 2, 6, 3, 4, 5, 6}.

Set the target value to remove: val = 6.

Create the linked list from the vector using createList.

Print the original list using printList.

Call removeElements method from the Solution class.

Print the updated list using printList.

Program:

```
#include <iostream>
#include <vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
    ListNode(int x, ListNode* next) : val(x), next(next) {}
};
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
```

```
        ListNode dummy(0, head); // Dummy node pointing to head
        ListNode* prev = &dummy; // Previous pointer starts at dummy
        ListNode* ptr = head;    // Current pointer starts at head
        while (ptr) {
            if (ptr->val == val) {
                prev->next = ptr->next; // Remove the node
            } else {
                prev = prev->next;      // Move prev forward
            }
            ptr = ptr->next;            // Always move ptr forward
        }
        return dummy.next; // New head (after dummy)
    }
};
ListNode* createList(const vector<int>& values) {
    if (values.empty()) return NULL;
    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;
    for (size_t i = 1; i < values.size(); ++i) {
        current->next = new ListNode(values[i]);
        current = current->next;
    }
    return head;
}
void printList(ListNode* head) {
    if (!head) {
        cout << "[]" << endl;
        return;
    }
    while (head) {
        cout << head->val;
        if (head->next) cout << " -> ";
        head = head->next;
    }
    cout << endl;
}
int main() {
    vector<int> input = {1, 2, 6, 3, 4, 5, 6}; // Example input
```

```
    int valToRemove = 6;
    ListNode* head = createList(input);
    cout << "Original list: ";
    printList(head);
    Solution solution;
    ListNode* updatedHead = solution.removeElements(head, valToRemove);
    cout << "List after removing " << valToRemove << ": ";
    printList(updatedHead);
    return 0;
}
```

Output:

Original list: 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6

List after removing 6: 1 -> 2 -> 3 -> 4 -> 5

LAB MODULE - V

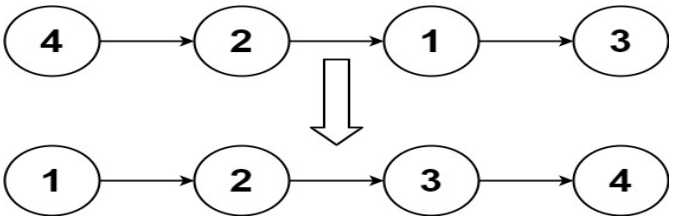
Linked List part 2

Program : 1

Aim : **Sort List**

Given the head of a linked list, return the list after sorting it in ascending order.

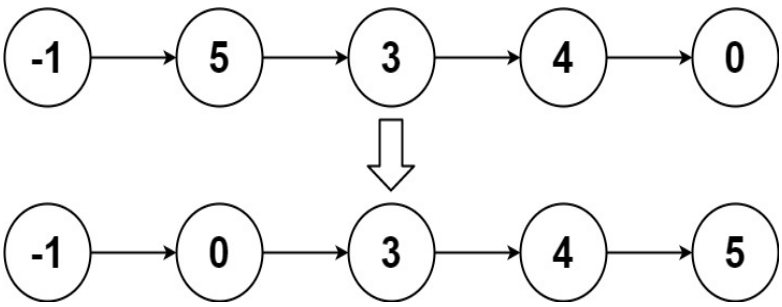
Example 1:



Input: head = [4,2,1,3]

Output: [1,2,3,4]

Example 2:



Input: head = [-1,5,3,4,0]

Output: [-1,0,3,4,5]

Constraints:

The number of nodes in the list is in the range [0, 5 * 10^4].

-10^5 <= Node.val <= 10^5

Algorithm:

Step 1: Define the Linked List Node Structure

Create a ListNode structure with:

int val: holds the node's value.

ListNode* next: pointer to the next node.

Step 2: Create a Helper Function to Build the List

Function: createList(vector<int>& vals)

If the vector is empty, return NULL.

Create a new ListNode for each element and link them.

Step 3: Print the Linked List

Function: printList(ListNode* head)

Traverse the list and print each node's value.

Step 4: Implement Merge Sort on Linked List

Class: Solution

Function: ListNode* sortList(ListNode* head)

Base case: If the list has 0 or 1 node, it's already sorted.

Use fast and slow pointers to find the middle node.

slow moves one step; fast moves two steps.

When fast reaches the end, slow will be at the midpoint.

Split the list into two halves.

Recursively sort each half using sortList.

Merge the sorted halves using merge.

Step 5: Merge Two Sorted Lists

Function: merge(ListNode* l1, ListNode* l2)

Create a dummy node to simplify edge cases.

Use a pointer cur to build the merged list.

Compare nodes from l1 and l2, attach the smaller to cur.

Append any remaining nodes from l1 or l2.

Step 6: Main Function Execution

Define a vector of integers: {4, 2, 1, 3}.

Convert it into a linked list using createList.

Print the original list.

Sort the list using sortList.

Print the sorted list.

Program:

```
#include <iostream>
#include <vector>
```

```
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == NULL || head->next == NULL)
            return head;
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast != NULL && fast->next != NULL) {
            slow = slow->next;
            fast = fast->next->next;
        }
        fast = slow->next;
        slow->next = NULL;
        return merge(sortList(head), sortList(fast));
    }
private:
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode dummy(0);
        ListNode* cur = &dummy;

        while (l1 != NULL && l2 != NULL) {
            if (l1->val < l2->val) {
                cur->next = l1;
                l1 = l1->next;
            } else {
                cur->next = l2;
                l2 = l2->next;
            }
            cur = cur->next;
        }
        cur->next = (l1 != NULL) ? l1 : l2;
        return dummy.next;
```

```
    }
};
ListNode* createList(const vector<int>& vals) {
    if (vals.empty()) return NULL;
    ListNode* head = new ListNode(vals[0]);
    ListNode* current = head;
    for (size_t i = 1; i < vals.size(); ++i) {
        current->next = new ListNode(vals[i]);
        current = current->next;
    }
    return head;
}
void printList(ListNode* head) {
    while (head != NULL) {
        cout << head->val;
        if (head->next != NULL) cout << " -> ";
        head = head->next;
    }
    cout << endl;
}
int main() {
    vector<int> input = {4, 2, 1, 3};
    ListNode* head = createList(input);
    cout << "Original List: ";
    printList(head);
    Solution solution;
    ListNode* sortedHead = solution.sortList(head);
    cout << "Sorted List: ";
    printList(sortedHead);
    return 0;
}
```

Output:-

Original List: 4 -> 2 -> 1 -> 3

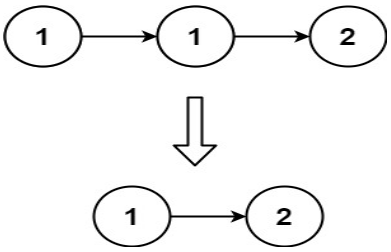
Sorted List: 1 -> 2 -> 3 -> 4

Program : 2

Aim : **Remove Duplicates from Sorted List**

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

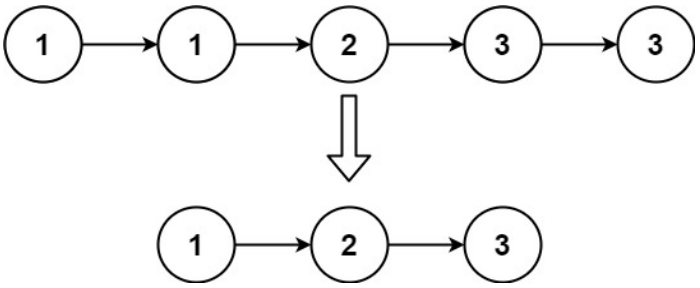
Example 1:



Input: head = [1,1,2]

Output: [1,2]

Example 2:



Input: head = [1,1,2,3,3]

Output: [1,2,3]

Constraints:

The number of nodes in the list is in the range [0, 300].

-100 <= Node.val <= 100

The list is guaranteed to be sorted in ascending order.

Algorithm:

Step 1: Define the Node Structure

Define a ListNode structure with:

int val to store the value.

ListNode* next to point to the next node.

Step 2: Build the Linked List from Vector

Function: createList(vector<int>& vals)

If the vector is empty, return NULL.

Initialize the head node with the first element.

Loop through the vector and create new nodes, linking them.

Step 3: Print the Linked List

Function: printList(ListNode* head)

Traverse the list from head to tail.

Print the value of each node.

Step 4: Remove Duplicates

Class: Solution

Function: deleteDuplicates(ListNode* head)

If the list is empty (head == NULL), return head.

Use a pointer curr to traverse the list.

While curr and curr->next are not NULL:

If curr->val == curr->next->val:

Store curr->next->next in a temporary pointer next.

Free memory of the duplicate node using delete.

Update curr->next = next to skip the duplicate.

Else:

Move curr to the next node.

Step 5: Main Function

Create a vector input {1, 1, 2, 3, 3}.

Convert it to a linked list using createList.

Print the original list.

Call deleteDuplicates to remove duplicates.

Print the modified list.

Program:

```
#include <iostream>
#include <vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == NULL) {
            return head;
        }
        ListNode* curr = head;
        while (curr != NULL && curr->next != NULL) {
            if (curr->val == curr->next->val) {
```

```
            ListNode* next = curr->next->next;
            delete curr->next;
            curr->next = next;
        } else {
            curr = curr->next;
        }
    }
    return head;
  }
};
ListNode* createList(const vector<int>& vals) {
  if (vals.empty()) return NULL;
  ListNode* head = new ListNode(vals[0]);
  ListNode* current = head;
  for (size_t i = 1; i < vals.size(); ++i) {
    current->next = new ListNode(vals[i]);
    current = current->next;
  }
  return head;
}
void printList(ListNode* head) {
  while (head != NULL) {
    cout << head->val;
    if (head->next != NULL) cout << " -> ";
    head = head->next;
  }
  cout << endl;
}
int main() {
  vector<int> input = {1, 1, 2, 3, 3};
  ListNode* head = createList(input);
  cout << "Original List: ";
  printList(head);
  Solution solution;
  ListNode* result = solution.deleteDuplicates(head);
  cout << "List after removing duplicates: ";
  printList(result);
  return 0;
```

}

Output:

Original List: 1 -> 1 -> 2 -> 3 -> 3

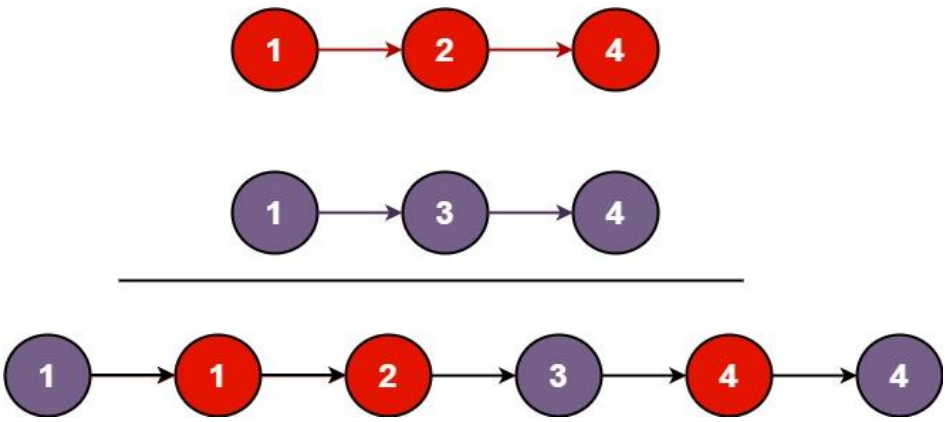
List after removing duplicates: 1 -> 2 -> 3

Program : 3

Aim : Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [ ], list2 = [0]

Output: [0]

Constraints:

The number of nodes in both lists is in the range [0, 50].

-100 <= Node.val <= 100

Both list1 and list2 are sorted in non-decreasing order.

Algorithm:

Step 1: Define Node Structure

Create a ListNode structure with:

An integer val to store the node's value.

A pointer next to the next node.

Step 2: Create Linked Lists

Function: createList(vector<int>& vals)

If input vector is empty, return NULL.

Otherwise, initialize head with the first value.

Loop through the rest of the vector, create nodes, and link them.

Step 3: Print Linked List

Function: printList(ListNode* head)

Traverse and print values.

Step 4: Merge Two Sorted Lists

Class: Solution

Function: mergeTwoLists(ListNode* l1, ListNode* l2)

If either l1 or l2 is NULL, return the non-null list.

Ensure l1 starts with the smaller value by swapping if needed.

Initialize:

result pointer to track start of merged list (same as l1)

temp pointer to help with linking

Traverse both lists until one becomes NULL:

If l1->val <= l2->val: move temp to l1, advance l1.

Else:

Set temp->next to l2

Swap l1 and l2 so l1 always points to smaller element.

After the loop, attach remaining nodes:

If l1 not empty, link it to temp->next

Else link l2 to temp->next

Return result (head of merged list).

Step 5: Main Function

Create vectors for input lists.

Convert vectors to linked lists using createList.

Print both lists.

Call mergeTwoLists and print the merged list.

Program:

```
#include <iostream>
#include <vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
```

```
        if (l1 == NULL) return l2;
        if (l2 == NULL) return l1;
        if (l1->val > l2->val) {
            swap(l1, l2);  // l1 should always be the smaller node
        }
        ListNode* result = l1;
        ListNode* temp = NULL;
        while (l1 != NULL && l2 != NULL) {
            if (l1->val <= l2->val) {
                temp = l1;
                l1 = l1->next;
            } else {
                temp->next = l2;
                swap(l1, l2);
            }
        }
        if (l1 != NULL) temp->next = l1;
        if (l2 != NULL) temp->next = l2;
        return result;
    }
};
ListNode* createList(const vector<int>& vals) {
    if (vals.empty()) return NULL;
    ListNode* head = new ListNode(vals[0]);
    ListNode* current = head;
    for (size_t i = 1; i < vals.size(); ++i) {
        current->next = new ListNode(vals[i]);
        current = current->next;
    }
    return head;
}
void printList(ListNode* head) {
    while (head != NULL) {
        cout << head->val;
        if (head->next != NULL) cout << " -> ";
        head = head->next;
    }
    cout << endl;
```

```
}
int main() {
    vector<int> input1 = {1, 2, 4};
    vector<int> input2 = {1, 3, 4};
    ListNode* list1 = createList(input1);
    ListNode* list2 = createList(input2);
    cout << "List 1: ";
    printList(list1);
    cout << "List 2: ";
    printList(list2);
    Solution solution;
    ListNode* mergedList = solution.mergeTwoLists(list1, list2);
    cout << "Merged List: ";
    printList(mergedList);
    return 0;
}
```

Output:

List 1: 1 -> 2 -> 4

List 2: 1 -> 3 -> 4

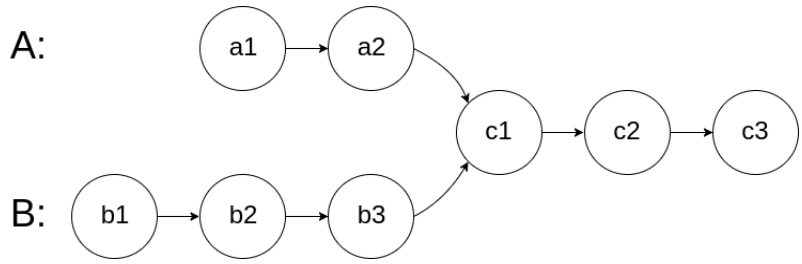
Merged List: 1 -> 1 -> 2 -> 3 -> 4 -> 4

Program : 4

Aim : Intersection of Two Linked Lists

Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

Note that the linked lists must retain their original structure after the function returns.

Custom Judge:

The inputs to the judge are given as follows (your program is not given these inputs):

intersectVal - The value of the node where the intersection occurs. This is 0 if there is no intersected node.

listA - The first linked list.

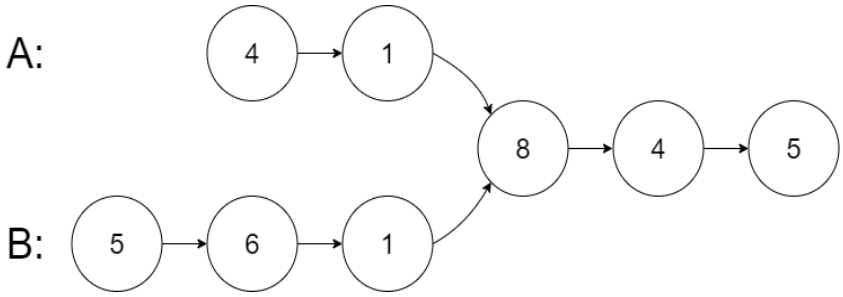
listB - The second linked list.

skipA - The number of nodes to skip ahead in listA (starting from the head) to get to the intersected node.

skipB - The number of nodes to skip ahead in listB (starting from the head) to get to the intersected node.

The judge will then create the linked structure based on these inputs and pass the two heads, headA and headB to your program. If you correctly return the intersected node, then your solution will be accepted.

Example 1:



Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3

Output: Intersected at '8'

Explanation: The intersected node's value is 8 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

- Note that the intersected node's value is not 1 because the nodes with value 1 in A and B (2nd node in A and 3rd node in B) are different node references. In other words, they point to two different locations in memory, while the nodes with value 8 in A and B (3rd node in A and 4th node in B) point to the same location in memory.

Constraints:

The number of nodes of listA is in the m.

The number of nodes of listB is in the n.

1 <= m, n <= 3 * 10^4

1 <= Node.val <= 10^5

0 <= skipA <= m

0 <= skipB <= n

intersectVal is 0 if listA and listB do not intersect.

intersectVal == listA[skipA] == listB[skipB] if listA and listB intersect.

Algorithm:

Step 1: Create Linked Lists

Use createList(vector<int> vals) to:

Convert a vector of integers into a singly linked list.

Returns the head of the created list.

Step 2: Simulate Intersection

Use attachTail(ListNode*& list, ListNode* tail, int skip) to:

Traverse the given list for skip number of nodes.

Attach tail at that position to simulate intersection.

Used for both listA and listB.

Step 3: Print Lists

Use printList(ListNode* head) to print the structure of listA and listB to visually confirm the structure.

Step 4: Find Intersection Node

Use getIntersectionNode(ListNode* headA, ListNode* headB) to:

Traverse headA, storing each node address in an unordered_map (hashmap).

Traverse headB, checking if any node address is already in the map.

If found, return that node as the intersection node.

If no match is found after traversal, return NULL.

Step 5: Output Result

In main():

Print whether an intersection is found.

If yes, output the intersecting node's value.

Program:

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;
struct ListNode {
  int val;
  ListNode *next;
  ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
  ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    unordered_map<ListNode*, int> vis;
    ListNode* temp = headA;
    while (temp != NULL) {
      vis[temp] = 1;
      temp = temp->next;
```

```
        }
        temp = headB;
        while (temp != NULL) {
            if (vis.find(temp) != vis.end()) {
                return temp; // Found intersection
            }
            temp = temp->next;
        }
        return NULL;
    }
};
ListNode* createList(const vector<int>& vals) {
    if (vals.empty()) return NULL;
    ListNode* head = new ListNode(vals[0]);
    ListNode* current = head;
    for (size_t i = 1; i < vals.size(); ++i) {
        current->next = new ListNode(vals[i]);
        current = current->next;
    }
    return head;
}
void printList(ListNode* head) {
    while (head != NULL) {
        cout << head->val;
        if (head->next != NULL) cout << " -> ";
        head = head->next;
    }
    cout << endl;
}
void attachTail(ListNode* &list, ListNode* tail, int skip) {
    ListNode* current = list;
    while (skip-- > 0 && current != NULL) {
        current = current->next;
    }
    if (current != NULL) {
        current->next = tail;
```

```
    }
}
int main() {
    vector<int> valsA = {4, 1};
    vector<int> valsB = {5, 6, 1};
    vector<int> common = {8, 4, 5};
    ListNode* listA = createList(valsA);
    ListNode* listB = createList(valsB);
    ListNode* commonTail = createList(common);
    attachTail(listA, commonTail, valsA.size() - 1); // skipA = 2
    attachTail(listB, commonTail, valsB.size() - 1); // skipB = 3
    cout << "List A: ";
    printList(listA);
    cout << "List B: ";
    printList(listB);
    Solution sol;
    ListNode* intersection = sol.getIntersectionNode(listA, listB);
    if (intersection != NULL)
        cout << "Intersected at '" << intersection->val << "'" << endl;
    else
        cout << "No intersection." << endl;
    return 0;
}
```

Output:

List A: 4 -> 1 -> 8 -> 4 -> 5

List B: 5 -> 6 -> 1 -> 8 -> 4 -> 5

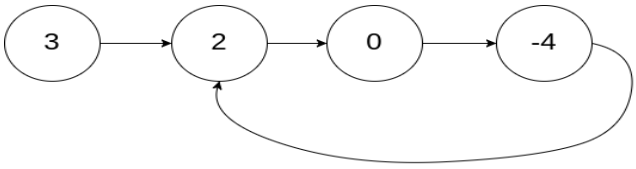
Intersected at '8'

Program : 5

Aim : Linked List Cycle

Given head, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter. Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1:

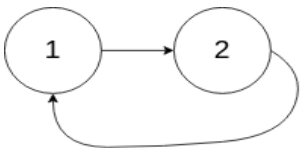


Input: head = [3,2,0,-4], pos = 1

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: head = [1,2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Constraints:

The number of the nodes in the list is in the range [0, 10^4].

-10^5 <= Node.val <= 10^5

pos is -1 or a valid index in the linked-list.

Algorithm:

Step 1: Create the Linked List:

Use createList(vals):

Initialize a head node with the first value.

Iterate through the remaining values.

Create a new node for each and connect it via the next pointer.

Step 2: Create a Cycle (if specified):

Use createCycle(head, pos):

Traverse the list to find the node at index pos. Store it as cycleStart.

Continue to the last node (tail) of the list.

Connect tail->next to cycleStart, forming a loop.

Step 3:Detect Cycle:

Use hasCycle(head):

Initialize two pointers, slow and fast, both at head.

Move slow one step at a time and fast two steps at a time.

If slow and fast meet at any point, a cycle is detected → return true.

If fast or fast->next becomes NULL, the list ends → return false.

Step 4:Output the Result:

Print "Cycle detected" if hasCycle returns true.

Otherwise, print "No cycle detected"

Program:

```
#include <iostream>
#include <vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
class Solution {
public:
    bool hasCycle(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if (fast == slow)
                return true;  // Cycle detected
        }
        return false;  // No cycle
    }
};
ListNode* createList(const vector<int>& vals) {
    if (vals.empty()) return NULL;
    ListNode* head = new ListNode(vals[0]);
    ListNode* current = head;
    for (size_t i = 1; i < vals.size(); ++i) {
        current->next = new ListNode(vals[i]);
        current = current->next;
    }
    return head;
}
void createCycle(ListNode* head, int pos) {
    if (pos < 0) return;
    ListNode* cycleStart = NULL;
    ListNode* tail = head;
    int index = 0;
```

```
    while (tail->next != NULL) {
        if (index == pos) {
            cycleStart = tail;
        }
        tail = tail->next;
        ++index;
    }
    tail->next = cycleStart;
}
int main() {
    vector<int> input = {3, 2, 0, -4};
    int pos = 1; // Position where the cycle begins (0-indexed)
    ListNode* head = createList(input);
    createCycle(head, pos);
    Solution sol;
    bool has_cycle = sol.hasCycle(head);
    if (has_cycle)
        cout << "Output: true (Cycle detected)" << endl;
    else
        cout << "Output: false (No cycle detected)" << endl;
    return 0;
}
```

Output:-

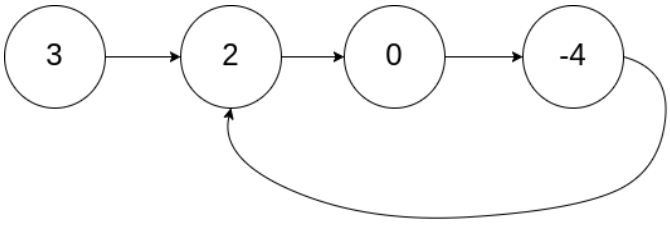
Output: true (Cycle detected)

Program : 6

Aim : **Linked List Cycle II**

Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter. Do not modify the linked list.

Example 1:



Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node.

Constraints:

The number of the nodes in the list is in the range [0, 10^4].

-10^5 <= Node.val <= 10^5

pos is -1 or a valid index in the linked-list.

Algorithm:

Step 1: Define the Node Structure: Define a ListNode struct that holds-

An integer val.

A pointer next to the next node.

Step 2: Implement Cycle Detection with Floyd's Algorithm

Create a Solution class with a method detectCycle(ListNode* head):

Initialize two pointers: slow and fast to head.

Move slow by 1 step and fast by 2 steps in each iteration.

If they meet, a cycle is detected.

To find the cycle's start:

Move slow back to head.

Move both slow and fast one step at a time.

When they meet again, it's the cycle's start node.

If fast or fast->next becomes nullptr, there's no cycle.

Step 3: Build the Linked List (with optional cycle):

Define a helper function createLinkedListWithCycle(values, pos):

Create nodes using values from the input vector.

Link them together to form a singly linked list.

Keep track of the node at index pos (cycle start).

At the end, if pos != -1, point the last node's next to the cycle start node, forming a cycle.

Step 4: Main Function to Test: In main():

Create a vector values = {3, 2, 0, -4} and pos = 1.

Use createLinkedListWithCycle to build the list.

Call detectCycle(head) from the Solution object.

If a node is returned, print its value — it's the cycle's start.

If nullptr is returned, print "No cycle detected".

Program:

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
```

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};
class Solution {
public:
    ListNode* detectCycle(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                slow = head;
                while (slow != fast) {
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow; // Start of the cycle
            }
        }
        return nullptr; // No cycle
    }
};
ListNode* createLinkedListWithCycle(vector<int> values, int pos) {
    if (values.empty()) return nullptr;
    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;
    ListNode* cycleStart = nullptr;
    for (size_t i = 1; i < values.size(); ++i) {
        current->next = new ListNode(values[i]);
        current = current->next;
        if (i == pos) {
            cycleStart = current;
        }
    }
    if (pos != -1) {
```

```
        current->next = (pos == 0) ? head : cycleStart;
    }
    return head;
}
int main() {
    vector<int> values = {3, 2, 0, -4};
    int pos = 1; // Tail connects to node at index 1
    ListNode* head = createLinkedListWithCycle(values, pos);
    Solution solution;
    ListNode* cycleNode = solution.detectCycle(head);
    if (cycleNode) {
        cout<<"Cycle detected at node with value:"<<cycleNode->val << endl;
    } else {
        cout << "No cycle detected." << endl;
    }
    return 0;
}
```

Output:-

Cycle detected at node with value: 2

Lab Module – VI

Stack & Queue

Evaluate Reverse Polish Notation:

https://leetcode.com/problems/evaluate-reverse-polish-notation/

You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.

Note that:

The valid operators are '+', '-', '*', and '/'.

Each operand may be an integer or another expression.

The division between two integers always truncates toward zero.

There will not be any division by zero.

The input represents a valid arithmetic expression in a reverse polish notation.

The answer and all the intermediate calculations can be represented in a 32-bit integer.

Example 1:

Input: tokens = ["2","1","+","3","*"]

Output: 9

Explanation: ((2 + 1) * 3) = 9

Constraints:

$1 <= tokens.length <= 10^4$

tokens[i] is either an operator: "+", "-", "*", or "/", or an integer in the range [-200, 200].

Algorithm:

1. Initialize a stack to store intermediate integer results.
2. For each token in the input list:
    - If the token is a number (including negative numbers):
        - Convert it to an integer and push it onto the stack.
    - Else, the token is an operator (+, -, *, /):
        - Pop the top two numbers from the stack:
            - Let operand2 = top (right-hand side)
            - Let operand1 = next (left-hand side)
        - Perform the operation operand1 op operand2.
        - Push the result back onto the stack.
3. After processing all tokens, the top of the stack is the final result.

Program :

```
#include <iostream>
#include <vector>
#include <string>
#include <stack>
#include <stdexcept>

using namespace std;

class Solution {
public:
    int evalRPN(vector<string>& tokens) {

        stack<int> numbers;

        for (const string& token : tokens) {

            if (token.size() > 1 || isdigit(token[0])) {

                numbers.push(stoi(token));
            } else {
                int operand2 = numbers.top();
                numbers.pop();
                int operand1 = numbers.top();
                numbers.pop();

                switch (token[0]) {
                    case '+':
                        numbers.push(operand1 + operand2);
                        break;
                    case '-':
                        numbers.push(operand1 - operand2);
                        break;
                    case '*':
                        numbers.push(operand1 * operand2);
                        break;
                    case '/':
                        numbers.push(operand1 / operand2);
                        break;
                }
            }
        }
        return numbers.top();
    }
};

int main() {
    cout << "Input RPN expression (space-separated tokens): ";
```

```
    string line;
    getline(cin, line);

    vector<string> tokens;
    string token;
    for (size_t i = 0; i < line.size(); ) {
        while (i < line.size() && isspace(line[i])){
            i++;
        }
        if (i >= line.size()) {
            break;
        }
        size_t j = i;
        while (j < line.size() && !isspace(line[j])) {
            j++;
        }
        token = line.substr(i, j - i);
        tokens.push_back(token);
        i = j;
    }

    Solution solution;
        int result = solution.evalRPN(tokens);
        cout << "Result: " << result << endl;
    return 0;
}
```

```
Input RPN expression (space-separated tokens): 2 1 + 3 *
Result: 9
```

Min Stack :

https://leetcode.com/problems/min-stack/description/

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

MinStack() initializes the stack object.

void push(int val) pushes the element val onto the stack.

void pop() removes the element on the top of the stack.

int top() gets the top element of the stack.

int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Example 1:

Input

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

Output

[null,null,null,null,-3,null,0,-2]

Explanation

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); // return -3

minStack.pop();

minStack.top(); // return 0

minStack.getMin(); // return -2

Constraints:

$-2^{31} <= val <= 2^{31} - 1$

Methods pop, top and getMin operations will always be called on non-empty stacks.

At most $3 * 10^4$ calls will be made to push, pop, top, and getMin.

Algorithm:

1. Initialize the Stack:

- Create two stacks:
  - mainStack: stores all values pushed by the user.
  - minValuesStack: keeps track of the **minimum value** at each level of the stack.
- Push INT_MAX to minValuesStack to handle edge cases.

2. Process Input:

- Read the number of operations.
- For each operation:
  - Read the operation name (MinStack, push, pop, top, getMin).
  - If the operation is push, read the integer value to be pushed.

3. Perform Operations:

- For each operation:
  - If MinStack:

        - Create a new instance of MinStack.
        - Output "null".
    - If push(val):
        - Push val to mainStack.
        - Push the smaller of val and the current minimum to minValuesStack.
        - Output "null".
    - If pop():
        - Pop the top element from both stacks.
        - Output "null".
    - If top():
        - Return the top of mainStack.
    - If getMin():
        - Return the top of minValuesStack.

4. Output Results:

- Print results of all operations in list format: [result1, result2, ...]

Program :

```
#include <iostream>
#include <stack>
#include <vector>
#include <string>
#include <limits.h>

using namespace std;

class MinStack {
public:
    MinStack() {
        minValuesStack.push(INT_MAX);
    }

    void push(int val) {
        mainStack.push(val);
        minValuesStack.push(min(val, minValuesStack.top()));
    }

    void pop() {
        mainStack.pop();
        minValuesStack.pop();
    }

    int top() {
        return mainStack.top();
    }

    int getMin() {
        return minValuesStack.top();
```

```
    }

private:
    stack<int> mainStack;
    stack<int> minValuesStack;
};

int main() {
    int n;
    cout << "Enter number of operations: ";
    cin >> n;

    vector<string> operations(n);
    vector<vector<int>> values(n);

    cout << "Enter operations and values (if needed):" << endl;
    for (int i = 0; i < n; ++i) {
        cin >> operations[i];
        if (operations[i] == "push") {
            int val;
            cin >> val;
            values[i].push_back(val);
        }
    }

    vector<string> output;
    MinStack* obj = nullptr;

    for (int i = 0; i < n; ++i) {
        if (operations[i] == "MinStack") {
            obj = new MinStack();
            output.push_back("null");
        } else if (operations[i] == "push") {
            obj->push(values[i][0]);
            output.push_back("null");
        } else if (operations[i] == "pop") {
            obj->pop();
            output.push_back("null");
        } else if (operations[i] == "top") {
            output.push_back(to_string(obj->top()));
        } else if (operations[i] == "getMin") {
            output.push_back(to_string(obj->getMin()));
        }
    }

    cout << "Output: [";
    for (size_t i = 0; i < output.size(); ++i) {
        cout << output[i];
```

```
        if (i != output.size() - 1)
            cout << ",";
    }
    cout << "]" << endl;

    delete obj;
    return 0;
}
```

Output :

```
Enter number of operations: 8
Enter operations and values (if needed):
MinStack
push -2
push 0
push -3
getMin
pop
top
getMin

Output: [null,null,null,null,-3,null,0,-2]
```

Daily Temperatures:

https://leetcode.com/problems/daily-temperatures/description/

Given an array of integer's temperatures represents the daily temperatures, return an array answer such that answer[i] is the number of days you have to wait after the i^{th} day to get a warmer temperature. If there is no future day for which this is possible, keep answer[i] == 0 instead.

Example 1:

Input: temperatures = [73,74,75,71,69,72,76,73]

Output: [1,1,4,2,1,1,0,0]

Constraints:

1 <= temperatures.length <= 10^5

30 <= temperatures[i] <= 100

Algorithm:

1. Initialize Structures:

- Let n be the number of days.
- Create a result list daysToWait of size n, initialized with 0s.
- Create an empty stack (indexStack) to keep track of the indices of unresolved days.

2. Traverse Temperatures:

- Loop through each day i from 0 to n - 1:
  - While the stack is not empty and the current temperature temperatures[i] is greater than the temperature at the index on the top of the stack:
    - Pop the index from the stack (let's call it previousDayIndex).
    - Calculate the difference: i - previousDayIndex → store this in daysToWait[previousDayIndex].
  - Push the current index i onto the stack.

3. Final Result:

- After processing all days, the remaining indices in the stack have no warmer day ahead, so their default value 0 stays unchanged.
- Return daysToWait.

Program :

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
class Solution {
public:
   vector<int> dailyTemperatures(vector<int>& temperatures) {
      int n = temperatures.size();
       vector<int> daysToWait(n);
      stack<int> indexStack;
      for (int i = 0; i < n; ++i) {

         while (!indexStack.empty() && temperatures[indexStack.top()] < temperatures[i])
         {
            int previousDayIndex = indexStack.top();
            daysToWait[previousDayIndex] = i - previousDayIndex;
            indexStack.pop();
         }

            indexStack.push(i);
      }
      return daysToWait;
   }
};
int main()
{
   int n;
   cout << "Enter number of days: ";
   cin >> n;

   vector<int> temperatures(n);
   cout << "Enter temperatures for each day:\n";
```

```
    for (int i = 0; i < n; ++i)
    {
        cin >> temperatures[i];
    }

    Solution solution;
    vector<int> result = solution.dailyTemperatures(temperatures);

    cout << "Days to wait for a warmer temperature:\n[";
    for (size_t i = 0; i < result.size(); ++i)
    {
        cout << result[i];
        if (i != result.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    return 0;
}
```

Output :

```
Enter number of days: 8
Enter temperatures for each day:
73 74 75 71 69 72 76 73
Days to wait for a warmer temperature:
[1, 1, 4, 2, 1, 1, 0, 0]
```

Largest Rectangle in Histogram:

https://leetcode.com/problems/largest-rectangle-in-histogram/description/

Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

Example 1:

Input: heights = [2,1,5,6,2,3]

Output: 10

Explanation: The above is a histogram where width of each bar is 1.

The largest rectangle is shown in the red area, which has an area = 10 units.

Example 2:

Input: heights = [2,4]

Output: 4

Constraints:

$1 <= heights.length <= 10^5$

$0 <= heights[i] <= 10^4$

Algorithm:

1. Initialize Data Structures:

- Let n be the number of bars (heights.size()).
- Create:
    - leftNearest: vector of size n, initialized to -1 — stores the index of the nearest smaller bar to the left.
    - rightNearest: vector of size n, initialized to n — stores the index of the nearest smaller bar to the right.
    - indexStack: empty stack for processing bar indices.

2. Traverse the Bars to Fill Left and Right Bounds:

Loop through each bar i from 0 to n - 1:

- While the stack is not empty and the current bar is smaller or equal to the bar at the top of the stack:
    - Update rightNearest[stack.top()] = i (because current bar is the next smaller to the right).
    - Pop the top index.
- If stack is not empty, set leftNearest[i] = stack.top() (top is the previous smaller to the left).
- Push current index i onto the stack.

3. Compute Maximum Area:

For each bar i:

- Width of rectangle = rightNearest[i] - leftNearest[i] - 1
- Height = heights[i]
- Area = height × width
- Update maxArea if this area is larger.

4. Return the Maximum Area

Program :

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

class Solution {
public:
   int largestRectangleArea(vector<int>& heights) {
      int maxArea = 0;
      int numBars = heights.size();
```

```
        stack<int> indexStack;
        vector<int> leftNearest(numBars, -1);
        vector<int> rightNearest(numBars, numBars);

        for (int i = 0; i < numBars; ++i) {
            while (!indexStack.empty() && heights[indexStack.top()] >= heights[i]) {
                rightNearest[indexStack.top()] = i;
                indexStack.pop();
            }
            if (!indexStack.empty()) leftNearest[i] = indexStack.top();
            indexStack.push(i);
        }

        for (int i = 0; i < numBars; ++i)
            maxArea = max(maxArea, heights[i] * (rightNearest[i] - leftNearest[i] - 1));

        return maxArea;
    }
};

int main() {
    int n;
    cout << "Enter number of bars in the histogram: ";
    cin >> n;

    vector<int> heights(n);
    cout << "Enter heights of the bars:\n";
    for (int i = 0; i < n; ++i) {
        cin >> heights[i];
    }

    Solution solution;
    int result = solution.largestRectangleArea(heights);

    cout << "Largest rectangle area in the histogram: " << result << endl;

    return 0;
}
```

Output :

```
Enter number of bars in the histogram: 6
Enter heights of the bars:
2 1 5 6 2 3
Largest rectangle area in the histogram: 10
```

Implement Stack using Queues:

https://leetcode.com/problems/implement-stack-using-queues/description/

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

void push(int x) Pushes element x to the top of the stack.

int pop() Removes the element on the top of the stack and returns it.

int top() Returns the element on the top of the stack.

boolean empty() Returns true if the stack is empty, false otherwise.

Notes:

You must use only standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.

Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

Input

["MyStack", "push", "push", "top", "pop", "empty"]

[[], [1], [2], [], [], []]

Output

[null, null, null, 2, 2, false]

Explanation

MyStack myStack = new MyStack();

myStack.push(1);

myStack.push(2);

myStack.top(); // return 2

myStack.pop(); // return 2

myStack.empty(); // return False

Algorithm:

1. Initialize Stack Structure

- Create a class MyStack with two queues:
    - mainQueue: holds the elements in stack order (top at front).
    - temporaryQueue: used during push to reorder elements.

2. push(x) Operation:

To push an element onto the stack:

- Enqueue x into temporaryQueue.
- Move all elements from mainQueue to temporaryQueue.
    - This ensures the new element is at the front (top of stack).
- Swap mainQueue and temporaryQueue.
    - mainQueue now has the correct order.

3. pop() Operation:

To remove the element on the top of the stack:

- Return and remove the front of mainQueue.

4. top() Operation:

To get the top element without removing it:

- Return the front of mainQueue.

5. empty() Operation:

To check if the stack is empty:

- Return whether mainQueue is empty.

6. Main Program Flow:

- Read n, the number of operations.
- For each operation:
    - Read the operation type (MyStack, push, pop, top, empty).
    - For push, read the value to push.
- Execute the operation accordingly.
- Store the output in a result list.

Program:

```
#include <iostream>
#include <queue>
#include <string>

using namespace std;

class MyStack {
public:
   MyStack() {}

   void push(int x) {
```

```
        temporaryQueue.push(x);
        while (!mainQueue.empty()) {
            temporaryQueue.push(mainQueue.front());
            mainQueue.pop();
        }
        swap(mainQueue, temporaryQueue);
    }

    int pop() {
        int topElement = mainQueue.front();
        mainQueue.pop();
        return topElement;
    }

    int top() {
        return mainQueue.front();
    }

    bool empty() {
        return mainQueue.empty();
    }

private:
    queue<int> mainQueue;
    queue<int> temporaryQueue;
};
int main() {
    int n;
    cout << "Enter number of operations: ";
    cin >> n;

    vector<string> operations(n);
    vector<vector<int>> values(n);

    cout << "Enter operations and arguments (e.g. push 10, pop):\n";
    for (int i = 0; i < n; ++i) {
        string op;
        cin >> op;
        operations[i] = op;
        if (op == "push") {
            int val;
            cin >> val;
            values[i].push_back(val);
        }
    }

    vector<string> output;
    MyStack* stack = nullptr;
```

```
    for (int i = 0; i < n; ++i) {
        string op = operations[i];
        if (op == "MyStack") {
            stack = new MyStack();
            output.push_back("null");
        } else if (op == "push") {
            stack->push(values[i][0]);
            output.push_back("null");
        } else if (op == "pop") {
            output.push_back(to_string(stack->pop()));
        } else if (op == "top") {
            output.push_back(to_string(stack->top()));
        } else if (op == "empty") {
            output.push_back(stack->empty() ? "true" : "false");
        }
    }

    cout << "Output: [";
    for (size_t i = 0; i < output.size(); ++i) {
        cout << output[i];
        if (i != output.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    delete stack;
    return 0;
}
```

Output :

```
Enter number of operations: 6
Enter operations and arguments (e.g. push 10, pop):
MyStack
push 1
push 2
top
pop
empty
Output: [null, null, null, 2, 2, false]
```

Implement Queue using Stacks:

https://leetcode.com/problems/implement-queue-using-stacks/description/

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

void push(int x) Pushes element x to the back of the queue.

int pop() Removes the element from the front of the queue and returns it.

int peek() Returns the element at the front of the queue.

boolean empty() Returns true if the queue is empty, false otherwise.

Notes:

You must use only standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.

Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

Input

["MyQueue", "push", "push", "peek", "pop", "empty"]

[[], [1], [2], [], [], []]

Output

[null, null, null, 1, 1, false]

Explanation

MyQueue myQueue = new MyQueue();

myQueue.push(1); // queue is: [1]

myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)

myQueue.peek(); // return 1

myQueue.pop(); // return 1, queue is [2]

myQueue.empty(); // return false

Constraints:

1 <= x <= 9

At most 100 calls will be made to push, pop, peek, and empty.

All the calls to pop and peek are valid.

Algorithm :

1. Initialize the Queue

- Create a class MyQueue with two stacks:
  - inputStack: stores newly pushed elements.
  - outputStack: used to reverse the order for FIFO behavior.

2. push(x) Operation

- Push x directly into inputStack.
- No element movement is needed at this point.

3. pop() Operation

- Ensure outputStack is ready using prepareOutputStack():
  - If outputStack is empty:
    - Transfer all elements from inputStack to outputStack.
    - This reverses the order to simulate a queue.
- Pop the top of outputStack (this is the front of the queue).

4. peek() Operation

- Same as pop(), but return the top of outputStack without removing it.

5. empty() Operation

- Return true if both stacks are empty.

6. Supporting Method: prepareOutputStack()

- Transfers all elements from inputStack to outputStack only if outputStack is empty.
- This ensures amortized O(1) performance.

7. Main Program Execution

- Read the number of operations n.
- For each operation (like MyQueue, push, pop, etc.):
  - Execute the method on the MyQueue object.
  - Store the result or "null" in the output list.

Program :

```
#include <iostream>
#include <vector>
#include <string>
#include <stack>
using namespace std;
```

```
class MyQueue {
public:

    MyQueue() {}
    void push(int x) {
        inputStack.push(x);
    }
    int pop() {
        prepareOutputStack();
        int element = outputStack.top();
        outputStack.pop();
        return element;
    }

    int peek() {
        prepareOutputStack();
        return outputStack.top();
    }
    bool empty() {
        return inputStack.empty() && outputStack.empty();
    }

private:
    stack<int> inputStack;
    stack<int> outputStack;
   void prepareOutputStack() {
        if (outputStack.empty()) {
            while (!inputStack.empty()) {
                outputStack.push(inputStack.top());
                inputStack.pop();
            }
        }
    }
};
int main() {
    int n;
    cout << "Enter number of operations: ";
    cin >> n;

    vector<string> operations(n);
    vector<vector<int>> values(n);

    cout << "Enter operations (e.g. MyQueue, push 1, pop, peek, empty):\n";
    for (int i = 0; i < n; ++i) {
        cin >> operations[i];
        if (operations[i] == "push") {
            int val;
            cin >> val;
```

```
            values[i].push_back(val);
        }
    }

    vector<string> output;
    MyQueue* queue = nullptr;

    for (int i = 0; i < n; ++i) {
        string op = operations[i];

        if (op == "MyQueue") {
            queue = new MyQueue();
            output.push_back("null");
        } else if (op == "push") {
            queue->push(values[i][0]);
            output.push_back("null");
        } else if (op == "pop") {
            output.push_back(to_string(queue->pop()));
        } else if (op == "peek") {
            output.push_back(to_string(queue->peek()));
        } else if (op == "empty") {
            output.push_back(queue->empty() ? "true" : "false");
        }
    }

    cout << "Output: [";
    for (size_t i = 0; i < output.size(); ++i) {
        cout << output[i];
        if (i != output.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    delete queue;
    return 0;
}
```

Output :

```
Enter number of operations: 6
Enter operations (e.g. MyQueue, push 1, pop, peek, empty):
MyQueue
push 1
push 2
peek
pop
empty
Output: [null, null, null, 1, 1, false]
```

Gas Station:

https://leetcode.com/problems/gas-station/description/

There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is guaranteed to be unique.

Example 1:

Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]

Output: 3

Explanation:

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

Example 2:

Input: gas = [2,3,4], cost = [3,4,3]

Output: -1

Explanation:

You can't start at station 0 or 1, as there is not enough gas to travel to the next station.

Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 0. Your tank = 4 - 3 + 2 = 3

Travel to station 1. Your tank = 3 - 3 + 3 = 3

You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.

Therefore, you can't travel around the circuit once no matter where you start.

Constraints:

n == gas.length == cost.length

$1 <= n <= 10^5$

$0 <= gas[i], cost[i] <= 10^4$

Algorithm :

1. Initialize Variables

- n: number of stations.
- start = n - 1: try to start from the last station.
- j = n - 1: a pointer to simulate station visits.
- tours = 0: number of stations visited.
- totalFuel = 0: track net fuel after subtracting cost.

2. Traverse Until Full Tour (**tours < n**)

- At each step:
  - Add net fuel from station j: gas[j] - cost[j] → update totalFuel.
  - Increase tours by 1.
  - Move j to the **next station** (circular): j = (j + 1) % n.

3. Handle Insufficient Fuel

- If totalFuel < 0:
  - Move start **backward** (i.e., try a different start point): start--.
  - Add net fuel from new start station.
  - Continue counting tours until tours == n.

4. Check Final Fuel

- If totalFuel < 0 after completing the loop → return -1 (not possible).
- Otherwise → return start (valid starting station).

Program :

```
#include <iostream>
#include <vector>

using namespace std;
class Solution {
public:
   int canCompleteCircuit(std::vector<int>& gas, std::vector<int>& cost) {
      int n = gas.size();
      int start = n - 1;
      int j = n - 1;
      int tours = 0;
      int totalFuel = 0;
      while (tours < n) {
         totalFuel += gas[j] - cost[j];
         tours++;
         j = (j + 1) % n;
         while (totalFuel < 0 && tours < n) {
            start--;
```

```
            totalFuel += gas[start] - cost[start];

            tours++;
          }
        }
        return totalFuel < 0 ? -1 : start;
    }
};
int main() {
    int n;
    cout << "Enter number of gas stations: ";
    cin >> n;

    vector<int> gas(n), cost(n);

    cout << "Enter gas available at each station:\n";
    for (int i = 0; i < n; ++i) {
        cin >> gas[i];
    }

    cout << "Enter cost to travel to next station from each station:\n";
    for (int i = 0; i < n; ++i) {
        cin >> cost[i];
    }

    Solution solution;
    int result = solution.canCompleteCircuit(gas, cost);

    cout << "Starting station index (or -1 if not possible): " << result << endl;

    return 0;
}
```

Output :

```
Enter number of gas stations: 5
Enter gas available at each station:
1 2 3 4 5
Enter cost to travel to next station from each station:
3 4 5 1 2
Starting station index (or -1 if not possible): 3
```

Sliding Window Maximum:

https://leetcode.com/problems/sliding-window-maximum/description/

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [3,3,5,5,6,7]

Explanation:

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Example 2:

Input: nums = [1], k = 1

Output: [1]

Constraints:

$1 <= nums.length <= 10^5$

$-10^4 <= nums[i] <= 10^4$

$1 <= k <= nums.length$

Algorithm:

1. Initialize Data Structures

- windowIndices: a deque that stores indices of array elements in decreasing order of values**.**
- maxValues: a vector to store the maximum of each window.

2. Traverse the Array

Loop through each index i in the array:

a. Remove Out-of-Window Indices

- If the index at the front of the deque is outside the current window (i - k >= front), remove it.

b. Maintain Decreasing Order in Deque

- While the deque is not empty and the current number nums[i] is greater than or equal to the number at the back of the deque:
  - Pop the back of the deque.
- This ensures the largest element's index is at the front of the deque.

c. Add Current Index

- Push the current index i to the back of the deque.

d. Record Maximum for the Window

- If the current index i >= k - 1, that means the window is valid:
  - The element at the front of the deque is the maximum → nums[deque.front()]
  - Add it to maxValues.

3. Return Result

- Return the maxValues vector which contains the maximum of each sliding window.

Program:

```
#include <iostream>
#include <vector>
#include <deque>
using namespace std;
class Solution {
public:
   vector<int> maxSlidingWindow(vector<int>& nums, int k) {
      deque<int> windowIndices;
      vector<int> maxValues;

      for (int i = 0; i < nums.size(); ++i) {
         if (!windowIndices.empty() && i - k >= windowIndices.front()) {
            windowIndices.pop_front();
         }
         while (!windowIndices.empty() && nums[windowIndices.back()] <= nums[i]) {
            windowIndices.pop_back();
         }
         windowIndices.push_back(i);
         if (i >= k - 1) {
            maxValues.emplace_back(nums[windowIndices.front()]);
         }
      }
```

```
        return maxValues;
    }
};
int main() {
    int n, k;
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> nums(n);
    cout << "Enter the elements:\n";
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    cout << "Enter window size k: ";
    cin >> k;

    Solution sol;
    vector<int> result = sol.maxSlidingWindow(nums, k);

    cout << "Maximums in each sliding window: [";
    for (size_t i = 0; i < result.size(); ++i) {
        cout << result[i];
        if (i != result.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    return 0;
}
```

Output :

```
Enter the number of elements: 8
Enter the elements:
1 3 -1 -3 5 3 6 7
Enter window size k: 3
Maximums in each sliding window: [3, 3, 5, 5, 6, 7]
```

Lab Module – VII

Hashing

Find Common Elements Between Two Arrays:

https://leetcode.com/problems/find-common-elements-between-two-arrays/description/

You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values:

answer1 : the number of indices i such that nums1[i] exists in nums2.

answer2 : the number of indices i such that nums2[i] exists in nums1.

Return [answer1,answer2].

Example 1:

Input: nums1 = [2,3,2], nums2 = [1,2]

Output: [2,1]

Example 2:

Input: nums1 = [4,3,2,3,1], nums2 = [2,2,5,2,3,6]

Output: [3,4]

Explanation:

The elements at indices 1, 2, and 3 in nums1 exist in nums2 as well. So answer1 is 3.

The elements at indices 0, 1, 3, and 4 in nums2 exist in nums1. So answer2 is 4.

Constraints:

n == nums1.length

m == nums2.length

1 <= n, m <= 100

1 <= nums1[i], nums2[i] <= 100

Algorithm :

1. Initialize Data Structures

- existenceNums1[101]: An array (size 101, assuming values between 0 and 100) to store whether a number exists in nums1[].
- existenceNums2[101]: An array to store whether a number exists in nums2[].
- intersectionCount[2]: A vector to store the two counts, initialized to zero.

2. Mark Elements Existence

- For each element in nums1[], set existenceNums1[num] = 1.
- For each element in nums2[], set existenceNums2[num] = 1.

3. Count Intersections

- For each element in nums1[], if it exists in nums2[] (i.e., existenceNums2[num] == 1), increment intersectionCount[0].
- For each element in nums2[], if it exists in nums1[] (i.e., existenceNums1[num] == 1), increment intersectionCount[1].

4. Return Result

- Return the vector intersectionCount[], which contains two values:
    1. The count of elements in nums1[] that are in nums2[].
    2. The count of elements in nums2[] that are in nums1[].

Program :

```
#include <iostream>
#include <vector>
using namespace std;
class Solution {
public:
    vector<int> findIntersectionValues(vector<int>& nums1, vector<int>& nums2)
{
        int existenceNums1[101]{};
        int existenceNums2[101]{};
        for (int num : nums1) {
            existenceNums1[num] = 1;
        }
        for (int num : nums2) {
            existenceNums2[num] = 1;
        }
        vector<int> intersectionCount(2);
        for (int num : nums1) {
            intersectionCount[0] += existenceNums2[num];
        }
        for (int num : nums2) {
            intersectionCount[1] += existenceNums1[num];
        }
        return intersectionCount;
    }
};
vector<int> readVector() {
    vector<int> vec;
    char ch;
    while (cin >> ch && ch != '[');
    int num;
    while (cin >> num) {
        vec.push_back(num);
        cin >> ch;
```

```
        if (ch == ']') break;
    }
    return vec;
}

int main() {
    cout << "Enter input (format: nums1 = [2,3,2], nums2 = [1,2]):" << endl;

    string dummy;
    cin >> dummy;
    vector<int> nums1 = readVector();

    cin >> dummy;
    cin >> dummy;
    vector<int> nums2 = readVector();

    Solution sol;
    vector<int> result = sol.findIntersectionValues(nums1, nums2);

    cout << "Output: [" << result[0] << "," << result[1] << "]" << endl;

    return 0;
}
```

Output :

```
Enter input (format: nums1 = [2,3,2], nums2 = [1,2]):
nums1 = [2,3,2], nums2 = [1,2]
Output: [2,1]
```

Contains Duplicate :

https://leetcode.com/problems/contains-duplicate/description/

Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1:

Input: nums = [1,2,3,1]

Output: true

Explanation:

The element 1 occurs at the indices 0 and 3.

Example 2:

Input: nums = [1,2,3,4]

Output: false

Explanation:

All elements are distinct.

Example 3:

Input: nums = [1,1,1,3,3,4,3,2,4,2]

Output: true

Constraints:

$1 <= nums.length <= 10^5$

$-10^9 <= nums[i] <= 10^9$

Algorithm:

1. Initialize a Set to Track Unique Elements

- Create an unordered set (numSet) which will store unique elements from the input array nums.
- The unordered set will automatically handle duplicates for us, as it only keeps unique values.

2. Insert Elements into the Set

- Convert the input vector nums to an unordered set numSet by initializing the set with the vector's elements:
  unordered_set<int> numSet(nums.begin(), nums.end());
- This step efficiently stores only unique elements from nums into the set.

3. Compare the Sizes

- Compare the size of the unordered set (numSet.size()) with the size of the input vector (nums.size()).
- If the size of the set is smaller than the size of the vector, it indicates that there were duplicates in the vector. This is because the set only retains unique elements.

4. Return the Result

- If the size of the set is less than the size of the vector, return true (indicating duplicates exist).
- If the size of the set is equal to the size of the vector, return false (indicating no duplicates).

Program:

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;
```

```cpp
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        unordered_set<int> numSet(nums.begin(), nums.end());
        return numSet.size() < nums.size();
    }
};
int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    vector<int> nums(n);
    cout << "Enter " << n << " integers:" << endl;
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    Solution sol;
    bool hasDuplicate = sol.containsDuplicate(nums);

    cout << "Output: " << (hasDuplicate ? "true" : "false") << endl;

    return 0;
}
```

Output :

```
Enter number of elements: 5
Enter 5 integers:
1 2 3 4 1
Output: true
```

Find All Duplicates in an Array:

https://leetcode.com/problems/find-all-duplicates-in-an-array/description/

Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears at most twice, return an array of all the integers that appears twice.

You must write an algorithm that runs in O(n) time and uses only constant auxiliary space, excluding the space needed to store the output.

Example 1:

Input: nums = [4,3,2,7,8,2,3,1]

Output: [2,3]

Example 2:

Input: nums = [1,1,2]

Output: [1]

Example 3:

Input: nums = [1]

Output: []

Constraints:

n == nums.length

$1 <= n <= 10^5$

1 <= nums[i] <= n

Each element in nums appears once or twice.

Algorithm :

1. Input the Array

- Accept an array of integers as input, e.g., nums[]. This array may contain duplicates.

2. Cycle Sorting to Place Numbers in Correct Indices

- The idea is to place each number at its correct index (i.e., for a number x, place it at index x-1).
- Cycle Sort:
    - Iterate over the array. For each element nums[i], check if nums[i] is already at its correct position (nums[i] == i + 1).
    - If it's not at its correct position, swap the element with the one at its correct position (i.e., swap nums[i] with nums[nums[i] - 1]).
    - Keep doing this until every element is in its correct position (or we find a duplicate during the process).

3. Identify Duplicates

- After sorting the array in the previous step:
    - Iterate over the array again.
    - For each index i, if nums[i] is not equal to i + 1, this means that the number at index i is a duplicate because the number should have been placed at index nums[i] - 1.
    - Add nums[i] to the result list of duplicates.

4. Return the Result

- After identifying all duplicates, return the list duplicates[].

Program :

```
#include <iostream>
```

```cpp
#include <vector>
#include <string>
#include <cctype>
using namespace std;

class Solution {
public:

    vector<int> findDuplicates(vector<int>& nums) {
        int size = nums.size();

        for (int i = 0; i < size; ++i) {

            while (nums[i] != nums[nums[i] - 1]) {
                swap(nums[i], nums[nums[i] - 1]);
            }
        }

        vector<int> duplicates;
        for (int i = 0; i < size; ++i) {
            if (nums[i] != i + 1) {
                duplicates.push_back(nums[i]);
            }
        }
        return duplicates;
    }
};
int main() {
    cout << "Input: ";

    vector<int> nums;
    int num;
    while (cin >> num) {
        nums.push_back(num);
        if (cin.peek() == '\n') break;
    }
    Solution sol;
    vector<int> result = sol.findDuplicates(nums);

    cout << "Output: [";
    for (size_t i = 0; i < result.size(); ++i) {
        cout << result[i];
        if (i != result.size() - 1) cout << ",";
    }
    cout << "]" << endl;

    return 0;
}
```

Output :

```
Input: 4 3 2 7 8 2 3 1
Output: [3,2]
```

Sort Characters By Frequency:

https://leetcode.com/problems/sort-characters-by-frequency/description/

Given a string s, sort it in decreasing order based on the frequency of the characters. The frequency of a character is the number of times it appears in the string.

Return the sorted string. If there are multiple answers, return any of them.

Example 1:

Input: s = "tree"

Output: "eert"

Explanation: 'e' appears twice while 'r' and 't' both appear once.

So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

Example 2:

Input: s = "cccaaa"

Output: "aaaccc"

Explanation: Both 'c' and 'a' appear three times, so both "cccaaa" and "aaaccc" are valid answers.

Note that "cacaca" is incorrect, as the same characters must be together.

Example 3:

Input: s = "Aabb"

Output: "bbAa"

Explanation: "bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

Constraints:

$1 <= s.length <= 5 * 10^5$

s consists of uppercase and lowercase English letters and digits.

Algorithm :

1. Count the Frequency of Each Character

- Create an unordered map frequencyMap where the key is the character and the value is the count of occurrences of that character.
- Loop through the string s and for each character, increment its count in the frequencyMap.

2. Create a List of Unique Characters

- After counting the frequencies, create a vector uniqueChars to store the unique characters present in the string. You can get these unique characters by iterating over the frequencyMap and pushing each key (character) into uniqueChars.

3. Sort the Unique Characters Based on Frequency

- Sort the uniqueChars vector in descending order based on the frequency of characters.
- Use the frequency map to compare frequencies of characters. The comparator for the sort function will compare the frequency of characters by accessing their counts in the frequencyMap.
- Sorting is done in such a way that characters with higher frequency appear first.

4. Build the Sorted Result String

- Initialize an empty string result.
- Iterate over the sorted uniqueChars vector and for each character ch, append it to result as many times as its frequency in the frequencyMap. This is done by concatenating a string string(frequencyMap[ch], ch) to result.

5. Return the Final String

- After constructing the result string with characters sorted by their frequency, return the result.

Program :

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    string frequencySort(string s) {
        unordered_map<char, int> frequencyMap;
        for (char ch : s) {
            ++frequencyMap[ch];
        }
        vector<char> uniqueChars;
        for (auto& keyValue : frequencyMap) {
            uniqueChars.push_back(keyValue.first);
        }
        sort(uniqueChars.begin(), uniqueChars.end(), [&](char a, char b) {
            return frequencyMap[a] > frequencyMap[b];
```

```
        });
        string result;
        for (char ch : uniqueChars) {
            result += string(frequencyMap[ch], ch);
        }
        return result;
    }
};
int main() {
    string s;
    cin >> s;

    Solution sol;
    string output = sol.frequencySort(s);

    cout <<output << endl;

    return 0;
}
```

Output :

```
tree
eert
```

Group Anagrams:

https://leetcode.com/problems/group-anagrams/

Given an array of strings strs, group the anagrams together. You can return the answer in any order.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Explanation:

There is no string in strs that can be rearranged to form "bat".

The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.

The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

Example 2:

Input: strs = [""]

Output: [[""]]

Example 3:

Input: strs = ["a"]

Output: [["a"]]

Constraints:

1 <= strs.length <= 10^4

0 <= strs[i].length <= 100

strs[i] consists of lowercase English letters.

Algorithm :

1. Initialize Data Structures

- Create an unordered map anagramGroups, where:
  - Key: A sorted version of each string (this is the key to group anagrams).
  - Value: A vector of strings (to store all anagrams corresponding to the same key).
- Create a vector groupedAnagrams to store the final result, which is a list of anagram groups.

2. Iterate Through the List of Strings

- For each string str in the input list strs, do the following:
  1. Sort the String: Sort the string str to create a canonical key (because anagrams will have the same sorted representation).
  2. Add to Map: Add the original string str to the vector corresponding to the sorted key in the anagramGroups map.

3. Build the Result

- Iterate through the anagramGroups map and add each group (the vector of anagrams) to the groupedAnagrams result.

4. Return the Result

- Return the groupedAnagrams, which is a vector of vectors containing anagrams grouped together.

Program :

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <sstream>
using namespace std;
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> anagramGroups;
```

```
        for (auto& str : strs) {
            string key = str;
            sort(key.begin(), key.end());
            anagramGroups[key].emplace_back(str);
        }
        vector<vector<string>> groupedAnagrams;
        for (auto& pair : anagramGroups) {
            groupedAnagrams.emplace_back(pair.second);
        }
        return groupedAnagrams;
    }
};
int main() {
    Solution solution;
    vector<string> strs;
    string inputLine;

    cout << "Enter strings separated by spaces :";
    getline(cin, inputLine);
    stringstream ss(inputLine);
    string word;
    while (ss >> word) {
        strs.push_back(word);
    }

    vector<vector<string>> result = solution.groupAnagrams(strs);

    cout << "[";
    for (size_t i = 0; i < result.size(); ++i) {
        cout << "[";
        for (size_t j = 0; j < result[i].size(); ++j) {
            cout << "\"" << result[i][j] << "\"";
            if (j < result[i].size() - 1) cout << ",";
        }
        cout << "]";
        if (i < result.size() - 1) cout << ",";
    }
    cout << "]" << endl;

    return 0;
}
```

Output :

```
Enter strings separated by spaces :eat tea tan ate nat bat
[["bat"],["tan","nat"],["eat","tea","ate"]]
```

Isomorphic Strings:

https://leetcode.com/problems/isomorphic-strings/description/

Given two strings s and t, determine if they are isomorphic.

Two strings s and t are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

Input: s = "egg", t = "add"

Output: true

Explanation:

The strings s and t can be made identical by:

Mapping 'e' to 'a'.

Mapping 'g' to 'd'.

Example 2:

Input: s = "foo", t = "bar"

Output: false

Explanation:

The strings s and t can not be made identical as 'o' needs to be mapped to both 'a' and 'r'.

Example 3:

Input: s = "paper", t = "title"

Output: true

Constraints:

$1 <= s.length <= 5 * 10^4$

t.length == s.length

s and t consist of any valid ascii character.

Algorithm :

1. Initialize Data Structures

- Use two arrays mappingS[256] and mappingT[256] to store the mappings of characters from s to t and t to s respectively. Each of these arrays is of size 256 (assuming the input strings contain only ASCII characters), and they are initialized to 0.
- These arrays will help track the mapping of characters between the two strings. If a character in s has already been mapped to a character in t, the mapping value will not change.

2. Iterate Through the Strings

- Iterate over each character in both strings s and t using an index i.
- For each character at index i in string s (denoted as charS) and string t (denoted as charT), check if their current mappings in the arrays mappingS and mappingT are consistent:
    - If mappingS[charS] != mappingT[charT], return false, as this indicates that the mappings are inconsistent (i.e., charS cannot map to charT in a one-to-one manner).
- If the mappings are consistent, update both mappings:
    - Set mappingS[charS] = i + 1 and mappingT[charT] = i + 1. This signifies that charS has been mapped to charT at position i.

3. Return the Result

- If no inconsistencies are found in the mappings after processing all characters, return true indicating that the strings are isomorphic.

Program :

```
#include <iostream>
#include <string>
using namespace std;
class Solution {
public:
    bool isIsomorphic(string s, string t)
    {
        int mappingS[256] = {0};
        int mappingT[256] = {0};
        int length = s.size();
        for (int i = 0; i < length; ++i)
        {
            char charS = s[i];
            char charT = t[i];
            if (mappingS[charS] != mappingT[charT])
            {
                return false;
            }
            mappingS[charS] = mappingT[charT] = i + 1;
        }
        return true;
    }
};
int main() {
    Solution solution;
    string s, t;
    cout << "Enter string s: ";
    cin >> s;

    cout << "Enter string t: ";
```

```
    cin >> t;
    if (s.length() != t.length()) {
        cout << "false" << endl;
        return 0;
    }

    bool result = solution.isIsomorphic(s, t);
    cout << (result ? "true" : "false") << endl;

    return 0;
}
```

Output :

```
Enter string s: egg
Enter string t: add
true
```

Module 8

Heap (Priority Queue)

Program1:

Last Stone Weight

Problem Statement:

You are given an array of integers stones where stones[i] is the weight of the i^{th} stone.

We are playing a game with the stones. On each turn, we choose the heaviest two stones and smash them together. Suppose the heaviest two stones have weights x and y with x <= y. The result of this smash is:

- If x == y, both stones are destroyed, and
- If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.

At the end of the game, there is at most one stone left.

Print *the weight of the last remaining stone*. If there are no stones left, Print 0.

Constraints:

- 1 <= stones.length <= 30
- 1 <= stones[i] <= 1000

Algorithm:

1.Initialize a max-heap

Create a priority_queue<int> named pq.
Insert all stone weights from the input vector into pq.

- This ensures you can always access the two heaviest stones in O(log n) time

2. **Process the stones repeatedly**
While there are at least two stones in pq:
a. Pop the heaviest stone and store as x = pq.top(); pq.pop();
b. Pop the next heaviest and store as y = pq.top(); pq.pop();
c. If x != y, compute the difference and push the result back: pq.push(x - y);

- This simulates smashing the two stones and, if they are unequal, reinserting the leftover piece

3. Return the result
After the loop finishes, either one stone remains or none.

- If pq is empty, return 0.
- Otherwise, return pq.top() — the last stone's weight

Implementation:

```
#include <iostream>
#include <vector>
#include<queue>
using namespace std;
int lastStoneWeight(vector<int>& stones) {
      priority_queue<int>pq;
      for(int i:stones)
      pq.push(i);
      while(pq.size()>1)
      {
        int x=pq.top();
        pq.pop();
        int y=pq.top();
        pq.pop();
        pq.push(x-y);

      }
        return pq.top();
    }
int main() {
int n
cin >>n;
vector<int> stones(n);
for (int i = 0; i < n; i++) {
cin >> stones[i];
}
cout<<lastStoneWeight(stones);
return 0;
}
```

Expected Results:

Input: n=6 stones = [2,7,4,1,8,1]

Output: 1

Program2:

Relative Ranks

Problem Statement:

You are given an integer array score of size n, where score[i] is the score of the i^{th} athlete in a competition. All the scores are guaranteed to be unique.

The athletes are placed based on their scores, where the 1^{st} place athlete has the highest score, the 2^{nd} place athlete has the 2^{nd} highest score, and so on. The placement of each athlete determines their rank:

- The 1^{st} place athlete's rank is "Gold Medal".
- The 2^{nd} place athlete's rank is "Silver Medal".
- The 3^{rd} place athlete's rank is "Bronze Medal".
- For the 4^{th} place to the n^{th} place athlete, their rank is their placement number (i.e., the x^{th} place athlete's rank is "x").

Print an array answer of size n where answer[i] is the rank of the i^{th} athlete.

Constraints:

- n == score.length
- $1 <= n <= 10^4$
- $0 <= score[i] <= 10^6$
- All the values in score are **unique**.

Algorithm:

1.Initialize a max-heap (pq)

- Create an empty priority_queue<int> pq;.
- Loop through each score[i] in the score vector and pq.push(score[i]);.
- This builds a max-heap of all scores—always letting you quickly access the highest score.

2. Build a map from score to rank

- Create an empty unordered_map<int,int> mp;.
- Initialize a rank counter int i = 1;.
- While pq is not empty:
    - Let topScore = pq.top(); pq.pop();.
    - Assign the current rank: mp[topScore] = i;
    - Increment i to assign the next rank to the next-highest score.
- After this loop:
    - mp[highest_score] == 1, mp[second_highest] == 2, etc.

3. Construct the result vector

- Create vector<string> res;.
- Loop through each original score s in score:
    - Retrieve r = mp[s];, the athlete's rank.
    - If r == 1, push "Gold Medal";
      else if r == 2, push "Silver Medal";
      else if r == 3, push "Bronze Medal";
      else push to_string(r);.
- This preserves the original order of athletes while converting ranks to the required strings.

4. Return the result

- Return the final **res** vector, with ranks labeled appropriately**.**

Implementation:

```
#include <iostream>
#include <vector>
#include<queue>
#include<unordered_map>
using namespace std;
    vector<string> findRelativeRanks(vector<int>& score) {
        priority_queue<int>pq;
        for(int i: score)
        pq.push(i);
        unordered_map<int,int>mp;
        int i=1;
        while(pq.size()!=0)
        {
            mp[pq.top()]=i;
            i++;
            pq.pop();
        }
        vector<string> res;
         for(int i: score){
            if(mp[i]==1)
            res.push_back("Gold Medal");
            else if(mp[i]==2)
            res.push_back("Silver Medal");
            else if(mp[i]==3)
             res.push_back("Bronze Medal");
             else
            res.push_back(to_string(mp[i]));
         }
        return res;
    }
int main() {
int n
cin >>n;
vector<int> score(n);
for (int i = 0; i < n; i++) {
cin >> score[i];
}
vector<string> ans= findRelativeRanks(score);
for(string s:ans){
    cout<<s<<" ";
}
return 0;
}
```

Expected Results:

Input: score = [10,3,8,9,4]
Output: ["Gold Medal","5","Bronze Medal","Silver Medal","4"]

Program3:

Take Gifts From the Richest Pile

Problem Statement:

You are given an integer array gifts denoting the number of gifts in various piles. Every second, you do the following:

- Choose the pile with the maximum number of gifts.
- If there is more than one pile with the maximum number of gifts, choose any.
- Reduce the number of gifts in the pile to the floor of the square root of the original number of gifts in the pile.

Print *the number of gifts remaining after* k *seconds.*

Constraints:

- 1 <= gifts.length <= 10^3
- 1 <= gifts[i] <= 10^9
- 1 <= k <= 10^3

Algorithm:

Initialize Max-Heap

- Create a priority_queue<int> named pq.
- Push all gift pile sizes from gifts into this heap.
    - This allows you to always extract the largest pile in O(log n) time per operation

2. Simulate Gifts Removal (Repeat for k Seconds)

- Loop k times (while (k--) { ... }):
    1. Extract the largest pile: int top = pq.top(); pq.pop();
    2. Compute floor(sqrt(top)) using sqrt() and wrap to long: long newSize = long(sqrt(top));
        - This calculates how many gifts remain in the pile after the rest are taken.
    3. Push the reduced pile size back into the heap: pq.push(newSize);

This mimics the problem's requirement: each second, the richest pile is partially taken, leaving behind its square-root quantity.

3. Sum Remaining Gifts

- Initialize long long sum = 0;
- While pq not empty:
    - Extract the top element and add to sum.

This yields the final count of gifts left in all piles.

4. Return the Result

- Return sum from the pickGifts function.

Implementation:

```
#include <iostream>
#include <vector>
#include<queue>
#include<cmath>
using namespace std;
long long pickGifts(vector<int>& gifts, int k) {
        priority_queue<int>pq;
        for (int g: gifts){
            pq.push(g);
        }
    while(k--)
    {
        pq.push(long(sqrt(pq.top())));
        pq.pop();
    }
   long long sum=0;
    while(pq.size()!=0){
        sum+=pq.top();
        pq.pop();
    }
      return sum;
 }
int main() {
int n,k;
cin >>n;
vector<int> gifts(n);
for (int i = 0; i < n; i++) {
cin >> gifts[i];
}
cin>>k;
cout<<pickGifts( gifts, k);
return 0;
}
```

Expected Results:

Input: gifts = [25,64,9,4,100], k = 4

Output: 29

Program4:

Seat Reservation Manager

Problem Statement: Design a system that manages the reservation state of n seats that are numbered from 1 to n.

Implement the SeatManager class:

- SeatManager(int n) Initializes a SeatManager object that will manage n seats numbered from 1 to n. All seats are initially available.
- int reserve() Fetches the smallest-numbered unreserved seat, reserves it, and returns its number.
- void unreserve(int seatNumber) Unreserves the seat with the given seatNumber.

Constraints:

- $1 <= n <= 10^5$
- $1 <= seatNumber <= n$
- For each call to reserve, it is guaranteed that there will be at least one unreserved seat.
- For each call to unreserve, it is guaranteed that seatNumber will be reserved.
- At most 10^5 calls **in total** will be made to reserve and unreserve.

Algorithm:

The approach we've taken is a combination of a Counter and Min-Heap strategy.

1. **Counter (p):** This keeps track of the latest continuous seat that's been reserved. For example, if seats 1, 2, and 3 are reserved and no unreservations have been made, last will be 3.
2. **Min-Heap (pq):** This is used to keep track of seats that have been unreserved and are out of the continuous sequence. For instance, if someone reserves seats 1, 2, and 3, and then unreserves seat 2, then seat 2 will be added to the min-heap.

The logic for the reserve and unreserve functions is as follows:

- **reserve:**
    - If the min-heap is empty, simply increment the last counter and return it.
    - If the min-heap has seats (i.e., there are unreserved seats), pop the smallest seat from the heap and return it.
- **unreserve:**
    - If the seat being unreserved is the last seat in the continuous sequence, decrement the last counter.
    - Otherwise, add the unreserved seat to the min-heap.

Implementation:

```
class SeatManager {
public:
      int p=0 ;
      priority_queue<int,vector<int>,greater<int>> pq;
    SeatManager(int n) {
       p =1;
    }
    int reserve() {
       if(pq.empty()){
          return p++;
       }
       else
       {
        int seat=pq.top();
            pq.pop();
        return seat;
       }
```

```
    }
    void unreserve(int seatNumber) {
        if(seatNumber==p)
            p--;
        else
        pq.push(seatNumber);
    }
};
```

Expected Results:
Input:
["SeatManager","reserve","unreserve","reserve","reserve","reserve","unreserve","reserve","unreserve","reserve","unreserve"]
[[4],[],[1],[],[],[],[2],[],[1],[],[2]]
Output:
[null,1,null,1,2,3,null,2,null,1,null]

Program5:

Minimum Amount of Time to Fill Cups

Problem Statement: You have a water dispenser that can dispense cold, warm, and hot water. Every second, you can either fill up 2 cups with different types of water, or 1 cup of any type of water.

You are given a 0-indexed integer array amount of length 3 where amount[0], amount[1], and amount[2] denote the number of cold, warm, and hot water cups you need to fill respectively. Return *the minimum number of seconds needed to fill up all the cups*.

Constraints:

- amount.length == 3
- 0 <= amount[i] <= 100

Algorithm:

Read input: Three integers representing amount[0], amount[1], amount[2].

Compute sum:

sum = amount[0] + amount[1] + amount[2]

Find the maximum:

mx = max(amount[0], amount[1], amount[2])

Determine minimum time:

- At most you can fill 2 cups per second, so the lower bound is ceil(sum / 2).
- On the other hand, if one type of drink has too many cups, you will need at least mx seconds (since you cannot fill more than 1 cup of the same type per second).
- Therefore, the answer is:

- time = max(mx, ceil(sum / 2))

Return/print time.

Implementation:

```
#include<iostream>
#include<vector>
using namespace std;
int fillCups(vector<int>& amount) {
    sort(amount.begin(),amount.end());
      int x=amount[0];
      int y=amount[1];
      int z=amount[2];
      int sum=x+y+z;
      if(x+y>z) return sum/2+sum%2;
      if(x==0 && y==0) return z;
      else return z;
    }
int main(){
    vector<int>amount(3);
    for(int i=0;i<3;i++)
        cin>>amount[i];
    cout<<fillCups(amount);
    return 0;
}
```

Expected Results:
Input: amount = [5,4,4]
Output: 7
Input: amount = [5,0,0]
Output: 5

Program6:

Reduce Array Size to The Half

Problem Statement:

You are given an integer array arr. You can choose a set of integers and remove all the occurrences of these integers in the array.

Return *the minimum size of the set so that at least half of the integers of the array are removed*.

Constraints:

- 2 <= arr.length <= 10^5
- arr.length is even.
- 1 <= arr[i] <= 10^5

Algorithm:

Steps:

1. **Count frequencies**
    - Initialize a hash map mp.
    - For each element x in arr, increment mp[x] by 1.
    - After this, mp stores frequency of each unique number.
2. **Store frequencies in a max-heap**
    - Initialize a max priority queue pq.
    - For each (key, freq) in mp, push freq into pq.
    - Now, the largest frequency is always at the top**.**
3. **Remove elements until array size ≤ n/2**
    - Let res = 0 (counter for unique elements removed).
    - Let curr_size = n (current array size).
    - Let target = n/2 (desired maximum size).
    - While curr_size > target:
        - Take f = pq.top() (most frequent element).
        - Remove it: curr_size -= f.
        - Pop it from pq.
        - Increment res++.
4. **Return result**
    - Return res (minimum number of unique elements removed).

Implementation:

```
#include<iostream>
#include<vector>
#include<unordered_map>
#include<queue>
using namespace std;
int minSetSize(vector<int>& arr) {
      unordered_map<int,int>mp;
      for(int i: arr){
        mp[i]+=1;
      }
      priority_queue<int>pq;
      for(auto i: mp)
      {
        pq.push(i.second);
      }
      int res=0;
      int n=arr.size();
```

```
        while(!pq.empty())
        {

            if(n-pq.top()>arr.size()/2)
            {
              n=n-pq.top();
              pq.pop();
              res++;
            }
            else{
              return ++res;
            }

        }
        return res;
    }
int main(){
    int n;
    cin>>n;
    vector<int>arr(n);
    for(int i=0;i<n;i++)
        cin>>arr[i];
    cout<<minSetSize(arr);
    return 0;
}
```

Expected Results:
Input: N=10
arr = [3,3,3,3,5,5,5,2,2,7]
Output: 2
Input: N=12
arr =[3,3,3,5,5,6,2,2,7,6,4,8]
Output: 3

Lab Module – 9
Greedy Approach to solve problems

Program1:

Maximum Sum With Exactly K Elements:

Problem Statement:

You are given a 0-indexed integer array nums and an integer k. Your task is to perform the following operation exactly k times in order to maximize your score:

1. Select an element m from nums.
2. Remove the selected element m from the array.
3. Add a new element with a value of m + 1 to the array.
4. Increase your score by m.

Return *the maximum score you can achieve after performing the operation exactly* k *times.*

Constraints:

- 1 <= nums.length <= 100
- 1 <= nums[i] <= 100
- 1 <= k <= 100

Algorithm:

Steps:

1. **Build a max-heap**
    - Initialize a max priority queue pq.
    - Push all elements of nums into pq.
2. **Repeat k times**
    - Extract the maximum element t = pq.top().
    - Add t to score.
    - Pop it from pq.
    - Push t+1 back into the heap (since that number increases after being chosen).
3. After k iterations, return score.

Implementation:

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;
 int maximizeSum(vector<int>& nums, int k) {
       priority_queue<int>pq;
       int score=0;
       for(int i: nums)
       {
        pq.push(i);
       }
       while(k--)
       {
        int t=pq.top();
        score+=t;
        pq.pop();
        pq.push(t+1);
       }
return score;
    }
int main(){
    int n,k;
    cin>>n;
```

```
    vector<int>nums(n);
    for(int i=0;i<n;i++)
        cin>>nums[i];
    cin>>k;
    cout<<maximizeSum(nums,k);
    return 0;
}
```

//Optimized

```
int maximizeSum(vector<int>& nums, int k) {
        sort(nums.begin(),nums.end());
        int ele=nums[nums.size()-1];
        return k*(2*ele+(k-1))/2;
    }
```

Expected Results:

Input: N=5 , nums = [1,2,3,4,5], k = 3
Output: 18

Program2:

Minimum Operations to Make the Array Increasing

Problem Statement: You are given an integer array nums (0-indexed). In one operation, you can choose an element of the array and increment it by 1.

- For example, if nums = [1,2,3], you can choose to increment nums[1] to make nums = [1,3,3].

Return *the minimum number of operations needed to make* nums *strictly increasing.*

An array nums is strictly increasing if nums[i] < nums[i+1] for all 0 <= i < nums.length - 1. An array of length 1 is trivially strictly increasing.

Constraints:

- 1 <= nums.length <= 5000
- 1 <= nums[i] <= 10^4

Algorithm:

Steps:

1. Initialize res = 0 (to count total operations).
2. Loop from i = 0 to n-2:
    - If nums[i] >= nums[i+1], then nums[i+1] is not strictly greater than nums[i].
    - We need to increase nums[i+1] to (nums[i] + 1) to maintain strictly increasing property.
    - The number of operations required = (nums[i] - nums[i+1] + 1).
    - Add this value to res.
    - Update nums[i+1] = nums[i] + 1.

3. Continue until the end of the array.
4. Return res.

Implementation:

```
#include<iostream>
#include<vector>
using namespace std;
  int minOperations(vector<int>& nums) {
        int res=0;
        int n=nums.size();
        for(int i=0;i<n-1;i++)
        {
            if(nums[i]>=nums[i+1]){
                res+=nums[i]-nums[i+1]+1;
                nums[i+1]+=nums[i]-nums[i+1]+1;
            }
        }
        return res;
    }
int main(){
    int n;
    cin>>n;
    vector<int>nums(n);
    for(int i=0;i<n;i++)
        cin>>nums[i];
    cout<<minOperations(nums);
    return 0;
}
```

Expected Results:

```
Input: nums = [1,5,2,4,1]
Output: 14
```

Program3: Assign Cookies

Problem Statement:

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

Each child i has a greed factor g[i], which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s[j]. If s[j] >= g[i], we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

Constraints:

- $1 <= g.length <= 3 * 10^4$
- $0 <= s.length <= 3 * 10^4$
- $1 <= g[i], s[j] <= 2^{31} - 1$

Algorithm:

Sort both arrays

- Sort g in ascending order (least greedy child first).
- Sort s in ascending order (smallest cookie first).

Initialize pointers and counter

- Set i = 0 (child index).
- Set j = 0 (cookie index).
- Set content = 0 (satisfied children count).

Match children with cookies

- While i < n and j < m:
    - If g[i] ≤ s[j] → child i can be satisfied with cookie j.
        - Increment content.
        - Move to the next child → i++.
    - Always move to the next cookie → j++.

End condition

- Stop when either all children are satisfied or all cookies are used.

Return result

- Return content as the maximum number of satisfied children.

Implementation:

```
#include<iostream>
#include<vector>
using namespace std;
   int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(),g.end());
        sort(s.begin(),s.end());
        int i=0,j=0;
        int content=0;
        while(i<g.size() && j<s.size())
        {
            if(g[i]<=s[j])
            {
                content++;
                i++;
            }
                j++;
        }
        return content;
    }
int main(){
    int n,m;
    cin>>n>>m;
    vector<int>g(n),s(m);
```

```
    for(int i=0;i<n;i++)
        cin>>g[i];
    for(int i=0;i<m;i++)
        cin>>s[i];
    cout<<findContentChildren(g,s);
    return 0;
}
```

Expected Results:
Input: n=3, m=5, g = [16,12,9] ,s = [1,2,3,11,7]
Output: 2

Program4: Non-overlapping Intervals

Problem Statement:

Given an array of intervals intervals where intervals[i] = [$start_i$, end_i], return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

Note that intervals which only touch at a point are non-overlapping. For example, [1,2] and [2, 3] are non-overlapping.

Constraints:

- 1 <= intervals.length <= 10^5
- intervals[i].length == 2
- $-5 * 10^4$ <= $start_i$ < end_i <= $5 * 10^4$

Algorithm:

1. **Sort intervals by start time**

   Sort all intervals in ascending order of start.

   Initialize variables

   end = intervals[0][1] (end of the first interval).

   ans = 0 (count of removed intervals).

2. **Traverse intervals**

   For each interval from i = 1 to n-1:

   a. If intervals[i][0] < end:

      i. Overlap detected.

      ii. Increment ans++ (remove one interval).

      iii. Keep the interval with the smaller end time (end = min(end, intervals[i][1])) → this ensures more room for future intervals.

   b. Else (no overlap):

i. Update end = intervals[i][1].

3. **Return answer**

After processing all intervals, return ans (minimum intervals removed).

Implementation:

```
#include<iostream>
#include<vector>
#include <algorithm>
using namespace std;
     int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        int ans = 0;
sort(intervals.begin(), intervals.end(), [](const vector<int>&a, const
vector<int> &b){
            return a[0]<b[0];
        });
        int n = intervals.size();
        int end;
        for(int i=0; i<n; i++){
            if(i==0) end = intervals[i][1];
            else if(intervals[i][0]<end) {
                ans++;
                end = min(end, intervals[i][1]);
            }
            else{
                end = intervals[i][1];
            }
        }
        return ans;
    }
int main(){
    vector<vector<int>> intervals(n,vector<int>(2));
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<2;j++)
        cin>>intervals[i][j];
    }
    cout<<eraseOverlapIntervals(intervals);
    return 0;
}
```

Expected Results:
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1

Program5: Boats to Save People

Problem Statement:

You are given an array people where people[i] is the weight of the i^{th} person, and an infinite number of boats where each boat can carry a maximum weight of limit. Each boat carries at most two people at the same time, provided the sum of the weight of those people is at most limit.

Return *the minimum number of boats to carry every given person*.

Constraints:

- $1 <= people.length <= 5 * 10^4$
- $1 <= people[i] <= limit <= 3 * 10^4$

Algorithm:

1. Sort the array people in ascending order.
   - This helps us efficiently pair the lightest and heaviest person.
2. Initialize pointers
   - i = 0 → lightest person (start of array).
   - j = n-1 → heaviest person (end of array).
   - ans = 0 → boat counter.
3. While i <= j (while people remain):
   - If the lightest (people[i]) + heaviest (people[j]) ≤ limit:
     - Put both in one boat → increment i++, decrement j--.
   - Else:
     - Put only the heaviest person (people[j]) in one boat → decrement j--.
   - Increment boat counter ans++ in both cases.
4. Return ans (minimum boats required).

Implementation:

```
#include<iostream>
#include<vector>
using namespace std;
   int numRescueBoats(vector<int>& people, int limit) {
        sort(people.begin(),people.end());
        int i=0,j=people.size()-1;
        int ans=0;
        while(i<=j){
           if(people[i]+people[j]<=limit)
           {
            i++;
            j--;
            ans++;
           }
           else{
            j--;
            ans++;
           }
        }
        return ans;
    }
int main(){
    int n,limit;
```

```
    cin>>n;
    vector<int>people(n);
    for(int i=0;i<n;i++)
        cin>>people[i];
        cin>>limit;
    cout<<numRescueBoats(people,limit);
    return 0;
}
```

Expected Results:
Input: people = [3,2,2,1], limit = 3
Output: 3

Program6: Lexicographically Smallest String After Substring Operation

Problem Statement:

Given a string s consisting of lowercase English letters. Perform the following operation:

- Select any non-empty substring then replace every letter of the substring with the preceding letter of the English alphabet. For example, 'b' is converted to 'a', and 'a' is converted to 'z'.

Return the lexicographically smallest string after performing the operation.

Constraints:

- $1 <= s.length <= 3 * 10^5$
- s consists of lowercase English letters

Algorithm:

- **Skip initial `'a'` characters:**
  - They are already smallest possible, so don't touch them.
  - Keep moving `i` forward until the first non-`'a'`.
- **If the whole string is `'a'`:**
  - Special case → change the **last character to `'z'`**.
- **Otherwise:**
  - From the first non-`'a'`, keep reducing characters (`ch → ch-1`) until you hit another `'a'` or end of string.
- **Return the final string.**

Implementation:

```
#include<iostream>
#include<vector>
using namespace std;
```

```
    string smallestString(string s) {
        int i=0;
        while(i<s.size() && s[i]=='a'){
            i++;
        }
        if(i==s.size())
        s[s.size()-1]='z';
        while(i<s.size()&&s[i]!='a')
        {
            --s[i++];
        }
        return s;
    }
int main(){
    string s;
    cin>>s;
    cout<<smallestString(s);
    return 0;
}
```

Expected Results:

Input: s = "aa"

Output: "az"

Input: s = "acbbc"

Output: "abaab"

Input: s = "leetcode"

Output: "kddsbncd"

Lab Module- X

1) Aim: **You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?**

Input: n = 2

Output: 2

Constraints:

1 <= n <= 45

Algorithm:

1. **Base Cases**:
   - If n = 1, there is only one way to climb (taking one step).
   - If n = 2, there are two ways: (1+1) or (2).
2. **Recursive Relation**:
   - The number of ways to reach step n is the sum of ways to reach (n-1) and (n-2), similar to the Fibonacci sequence.
3. **Dynamic Programming Approach**:
   - Use an array dp[] to store results of subproblems.
   - Initialize dp[1] = 1 and dp[2] = 2.
   - Iterate from 3 to n and calculate dp[i] = dp[i-1] + dp[i-2].
   - Return dp[n].

Code:

```
#include <iostream>
#include <vector>

using namespace std;

int climbStairs(int n) {
    if (n == 1) return 1;

    vector<int> dp(n + 1);
    dp[1] = 1;
    dp[2] = 2;


    for (int i = 3; i <= n; ++i) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[n];
```

```
}

int main() {
    int n;
    cin>>n;
    cout<<climbStairs(n) << endl;
        return 0;
    }
```

2) Aim: You have intercepted a secret message encoded as a string of numbers. The message is decoded via the following mapping:

"1" -> 'A' "2" -> 'B' ... "25" -> 'Y' "26" -> 'Z'

However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes ("2" and "5" vs "25").

For example, "11106" can be decoded into:

"AAJF" with the grouping (1, 1, 10, 6)

"KJF" with the grouping (11, 10, 6)

The grouping (1, 11, 06) is invalid because "06" is not a valid code (only "6" is valid).

Note: there may be strings that are impossible to decode.

Input: s = "12"

Output: 2

Constraints:

1 <= s.length <= 100

s contains only digits and may contain leading zero(s).

Algorithm:

1. **Base Cases**:
    - If the string is empty or starts with '0', there are **zero** ways to decode it.
2. **Dynamic Programming Approach**:
    - Create an array dp[] where dp[i] represents the number of ways to decode the substring of length i.
    - Initialize dp[0] = 1 (empty string) and dp[1] = 1 if the first character is valid.
    - Iterate through the string from index 2 to n:
        - If s[i-1] is non-zero, add dp[i-1].

- If s[i-2]s[i-1] forms a valid two-digit number (between "10" and "26"), add dp[i-2].

Code:

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int numDecodings(string s) {
    int n = s.size();
    if (s.empty() || s[0] == '0') return 0;

    vector<int> dp(n + 1, 0);
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i <= n; ++i) {
        if (s[i - 1] != '0')
            dp[i] += dp[i - 1];

        int twoDigit = stoi(s.substr(i - 2, 2));
        if (twoDigit >= 10 && twoDigit <= 26)
            dp[i] += dp[i - 2];
    }

    return dp[n];
}

int main() {
    string s;
    cin >> s;
    cout << numDecodings(s) << endl;
}
```

3) Aim: **You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.**

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Input: nums = [1,2,3,1]

Output: 4

Constraints:

1 <= nums.length <= 100

0 <= nums[i] <= 400

Algorithm:

1. **Base Cases**:
   - If there's only one house, rob it.
   - If there are two houses, rob the one with more money.
2. **Dynamic Programming Approach**:
   - Create an array dp[] where dp[i] represents the maximum money that can be robbed up to house i.
   - dp[i] = max(dp[i-1], dp[i-2] + nums[i]), meaning:
     - Either **skip the current house** and keep the previous max (dp[i-1]).
     - Or **rob the current house** plus the best money from skipping one house (dp[i-2] + nums[i]).
   - Return dp[n-1] as the maximum money that can be robbed.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

int rob(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;
    if (n == 1) return nums[0];

    vector<int> dp(n);
    dp[0] = nums[0];
    dp[1] = max(nums[0], nums[1]);
```

```
    for (int i = 2; i < n; ++i) {
        dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]);
    }

    return dp[n - 1];
}

int main() {
    int n;
    cin >> n;
    vector<int> nums(n);

    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    cout << rob(nums) << endl;
}
```

5) Aim: Given an integer array nums, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.

Input: nums = [1,5,11,5]

Output: true

Constraints:

1 <= nums.length <= 200

1 <= nums[i] <= 100

Algorithm:

1. **Calculate Total Sum**:
    - If the sum of the array is **odd**, it **cannot** be divided into two equal subsets, return false.
2. **Subset Sum Problem**:
    - Convert the problem to finding a subset with sum = totalSum / 2.
    - Use **dynamic programming** (dp[i] represents whether subset sum i is possible).
    - Initialize dp[0] = true (sum of 0 is always achievable).
    - Iterate through numbers and update possible subset sums.

Code:

```cpp
#include <iostream>
#include <vector>

using namespace std;

bool canPartition(vector<int>& nums) {
    int totalSum = 0;
    for (int num : nums) totalSum += num;

    if (totalSum % 2 != 0) return false; // If sum is odd, cannot split
equally

    int target = totalSum / 2;
    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int num : nums) {
        for (int j = target; j >= num; --j) {
            dp[j] = dp[j] || dp[j - num];
        }
    }

    return dp[target];
}

int main() {
    int n;
    cin >> n;
    vector<int> nums(n);

    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    cout << (canPartition(nums) ? "true" : "false") << endl;
}
```

6) Aim: **A frog is crossing a river. The river is divided into some number of units, and at each unit, there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.**

Given a list of stones positions (in units) in sorted ascending order, determine if the frog can cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be 1 unit. If the frog's last jump was k units, its next jump must be either k - 1, k, or k + 1 units. The frog can only jump in the forward direction.

Input: stones = [0,1,3,5,6,8,12,17]

Output: true

Constraints:

2 <= stones.length <= 2000

0 <= stones[i] <= 231 - 1

stones[0] == 0

stones is sorted in a strictly increasing order.

Algorithm:

1. **Base Cases**:
   - If the second stone isn't 1 unit away (stones[1] != 1), return false, as the frog's first jump must be exactly **1**.
2. **Dynamic Programming with Hash Map**:
   - Use a map dp where dp[position] is a **set** storing all valid jump lengths the frog can take from that position.
   - Start at stones[0] and add {1} to dp[1] since the first jump is fixed.
   - Iterate through stones, updating possible jumps for each reachable position.
   - If the last stone contains reachable jump lengths, return true.

Code:

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <unordered_set>

using namespace std;

bool canCross(vector<int>& stones) {
    int n = stones.size();
    if (stones[1] != 1) return false; // First jump must be exactly 1

    unordered_set<int> stoneSet(stones.begin(), stones.end()); // Quick lookup
```

```
    unordered_map<int, unordered_set<int>> dp; // Stores possible jump sizes
for each stone
    dp[0].insert(0);

    for (int pos : stones) {
        for (int step : dp[pos]) { // Iterate over valid jump lengths
            for (int nextStep : {step - 1, step, step + 1}) { // Can jump k-1,
k, k+1
                if (nextStep > 0) { // Steps must be positive
                    int nextPos = pos + nextStep;
                    if (nextPos == stones.back()) return true; // Frog reaches
last stone
                    if (stoneSet.count(nextPos)) dp[nextPos].insert(nextStep);
// Only add if stone exists
                }
            }
        }
    }

    return false;
}

int main() {
    int n;
    cin >> n;
    vector<int> stones(n);

    for (int i = 0; i < n; ++i) {
        cin >> stones[i];
    }

    cout << (canCross(stones) ? "true" : "false") << endl;
}
```

7) Aim: **There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time**.

Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner. The test cases are generated so that the answer will be less than or equal to 2 * 109.

Input: m = 3, n = 7

Output: 28

Constraints:

1 <= m, n <= 100

Algorithm:

1. Create a dp table where dp[i][j] represents the number of ways to reach cell (i, j).
2. Initialize dp[0][j] = 1 and dp[i][0] = 1 since there's only one way to reach the first row and first column.
3. Use the recurrence relation:

   $$dp[i][j]=dp[i-1][j]+dp[i][j-1]dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

   This means the number of ways to reach (i, j) is the sum of ways to reach from the top and left.

4. Return dp[m-1][n-1].

Code:

```cpp
#include <iostream>
#include <vector>

using namespace std;

int uniquePaths(int m, int n) {
    vector<vector<int>> dp(m, vector<int>(n, 1));

    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }

    return dp[m-1][n-1];
}

int main() {
    int m, n;
    cin >> m >> n;
    cout << uniquePaths(m, n) << endl;
}
```

8) Aim: Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Input: grid = [[1,3,1],[1,5,1],[4,2,1]]

Output: 7

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 200

0 <= grid[i][j] <= 200

Algorithm:

1. **Define** dp[i][j] as the minimum path sum to reach cell (i, j).
2. **Base Case:**
   - dp[0][0] = grid[0][0] (starting point).
   - First row: dp[0][j] = dp[0][j-1] + grid[0][j] (can only move right).
   - First column: dp[i][0] = dp[i-1][0] + grid[i][0] (can only move down).
3. **Transition Formula:**
   - dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]
   - This ensures that the minimum sum is carried forward from either the top or left cell.
4. **Return** dp[m-1][n-1] as the final answer.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<int>> dp(m, vector<int>(n));

    dp[0][0] = grid[0][0];

    for (int j = 1; j < n; ++j) {
        dp[0][j] = dp[0][j-1] + grid[0][j];
    }

    for (int i = 1; i < m; ++i) {
        dp[i][0] = dp[i-1][0] + grid[i][0];
    }
```

```
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
        }
    }

    return dp[m-1][n-1];
}

int main() {
    int m, n;
    cin >> m >> n;
    vector<vector<int>> grid(m, vector<int>(n));

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> grid[i][j];
        }
    }

    cout << minPathSum(grid) << endl;
}
```

9) Aim: There are N items, numbered 1,2,…,N. For each i (1≤i≤N), Item i has a weight of wi and a value of vi. Taro has decided to choose some of the N items and carry them home in a knapsack. The capacity of the knapsack is W, which means that the sum of the weights of items taken must be at most W.

Find the maximum possible sum of the values of items that Taro takes home.

Constraints:

All values in input are integers.

$1 \le N \le 100$

$1 \le W \le 10^5$

$1 \le w_i \le W$

$1 \le v_i \le 10^9$

Input

3 8

3 30

4 50

5 60

Output

90

Algorithm:

1. **Define** dp[i][w] as the maximum value achievable using the first i items with a total weight limit w.
2. **Base Case:**
   - If i == 0 or w == 0, then dp[i][w] = 0 (no items or no capacity).
3. **Transition Formula:**
   - If the current item's weight wi is **less than or equal** to w, we have two choices:
     - **Include the item**: dp[i][w] = max(dp[i-1][w], dp[i-1][w-wi] + vi)
     - **Exclude the item**: dp[i][w] = dp[i-1][w]
   - Otherwise, we **must exclude** the item.

Code:

10) Aim: Given an integer array nums, return the length of the longest strictly increasing subsequence.

Input: nums = [10,9,2,5,3,7,101,18]

Output: 4

Constraints:

1 <= nums.length <= 2500

-10^4 <= nums[i] <= 10^4

Code:

```
#include <iostream>
#include <vector>

using namespace std;
```

```
int knapsack(int N, int W, vector<int>& weights, vector<int>& values) {
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= N; ++i) {
        for (int w = 0; w <= W; ++w) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] +
values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[N][W];
}

int main() {
    int N, W;
    cin >> N >> W;
    vector<int> weights(N), values(N);

    for (int i = 0; i < N; ++i) {
        cin >> weights[i] >> values[i];
    }

    cout << knapsack(N, W, weights, values) << endl;
}
```

11) Aim: **Given a set of distinct positive integers nums, return the largest subset answer such that every pair (answer[i], answer[j]) of elements in this subset satisfies:**

answer[i] % answer[j] == 0, or

answer[j] % answer[i] == 0

If there are multiple solutions, return any of them.

Input: nums = [1,2,3]

Output: [1,2]

Constraints:

1 <= nums.length <= 1000

1 <= nums[i] <= 2 * 109

All the integers in nums are unique.

Algorithm:

1. **Sort the Array**: Sorting helps ensure that if a divides b, then a appears before b.
2. **Define** dp[i]: dp[i] stores the largest divisible subset ending at index i.
3. **Transition Formula**:
    - For each nums[i], check all previous elements nums[j] (j < i).
    - If nums[i] % nums[j] == 0, extend dp[j] to include nums[i].
    - Keep track of the largest subset found.
4. **Retrieve the Largest Subset**.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<int> largestDivisibleSubset(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return {};

    sort(nums.begin(), nums.end());
    vector<int> dp(n, 1), prev(n, -1);
    int maxSize = 1, maxIndex = 0;

    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[i] % nums[j] == 0 && dp[i] < dp[j] + 1) {
                dp[i] = dp[j] + 1;
                prev[i] = j;
            }
        }
        if (dp[i] > maxSize) {
            maxSize = dp[i];
            maxIndex = i;
        }
    }

    vector<int> result;
    for (int i = maxIndex; i != -1; i = prev[i]) {
        result.push_back(nums[i]);
    }

    reverse(result.begin(), result.end());
```

```
    return result;
}

int main() {
    int n;
    cin >> n;
    vector<int> nums(n);

    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    vector<int> result = largestDivisibleSubset(nums);
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;
}
```

Lab Module- XI

1) Aim: Given the root of a binary tree, return the preorder traversal of its nodes' values.

Input: root = [1,null,2,3]

Output: [1,2,3]

Constraints:

The number of nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

Algorithm:

Preorder traversal of a binary tree follows the **Root → Left → Right** order. This means:

1. Visit the root node first.
2. Recursively traverse the left subtree.
3. Recursively traverse the right subtree.

Code:

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

void preorderTraversal(TreeNode* root, vector<int>& result) {
    if (!root) return;
    result.push_back(root->val);
    preorderTraversal(root->left, result);
    preorderTraversal(root->right, result);
}

vector<int> preorder(TreeNode* root) {
    vector<int> result;
    preorderTraversal(root, result);
    return result;
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    vector<int> output = preorder(root);
    for (int val : output) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

2) Aim: Given the root of a binary tree, return the inorder traversal of its nodes' values.

Input: root = [1,null,2,3]

Output: [1,3,2]

Constraints:

The number of nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

Algorithm:

Inorder traversal of a binary tree follows the **Left → Root → Right** order. This means:

1. Recursively traverse the left subtree.
2. Visit the root node.
3. Recursively traverse the right subtree.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

void inorderTraversal(TreeNode* root, vector<int>& result) {
    if (!root) return;
    inorderTraversal(root->left, result);
    result.push_back(root->val);
    inorderTraversal(root->right, result);
}

vector<int> inorder(TreeNode* root) {
    vector<int> result;
    inorderTraversal(root, result);
    return result;
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    vector<int> output = inorder(root);
    for (int val : output) {
```

```
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

3)Aim: Given the root of a binary tree, return the postorder traversal of its nodes' values.

Input: root = [1,null,2,3]

Output: [3,2,1]

Constraints:

The number of the nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

Algorithm:

Postorder traversal of a binary tree follows the **Left → Right → Root** order. This means:

1. Recursively traverse the left subtree.
2. Recursively traverse the right subtree.
3. Visit the root node.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

void postorderTraversal(TreeNode* root, vector<int>& result) {
    if (!root) return;
    postorderTraversal(root->left, result);
    postorderTraversal(root->right, result);
```

```
    result.push_back(root->val);
}

vector<int> postorder(TreeNode* root) {
    vector<int> result;
    postorderTraversal(root, result);
    return result;
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    vector<int> output = postorder(root);
    for (int val : output) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

4) Aim: Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Input: root = [3,9,20,null,null,15,7]

Output: 3

Constraints:

The number of nodes in the tree is in the range [0, 104].

-100 <= Node.val <= 100

Algorithm:

1. **Base Case:** If the tree is empty (root == nullptr), return 0.
2. **Recursive Case:**
    - Compute the depth of the left subtree.
    - Compute the depth of the right subtree.
    - The depth of the current node is max(leftDepth, rightDepth) + 1.

Code:

```cpp
#include <iostream>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

int maxDepth(TreeNode* root) {
    if (!root) return 0;
    return max(maxDepth(root->left), maxDepth(root->right)) + 1;
}

int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    cout << maxDepth(root) << endl;
    return 0;
}
```

5) Aim: Given the root of a binary tree, return the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root. The length of a path between two nodes is represented by the number of edges between them.

Input: root = [1,2,3,4,5]

Output: 3

Constraints:

The number of nodes in the tree is in the range [1, 104].

-100 <= Node.val <= 100

Algorithm:

1. **Define a helper function** that computes the height of each subtree.
2. **Track the maximum diameter** while computing the height.
3. **Return the maximum diameter** found during traversal.

Code:

```cpp
#include <iostream>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

int diameterHelper(TreeNode* root, int& diameter) {
    if (!root) return 0;

    int leftHeight = diameterHelper(root->left, diameter);
    int rightHeight = diameterHelper(root->right, diameter);

    diameter = max(diameter, leftHeight + rightHeight);

    return max(leftHeight, rightHeight) + 1;
}

int diameterOfBinaryTree(TreeNode* root) {
    int diameter = 0;
    diameterHelper(root, diameter);
    return diameter;
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);

    cout << diameterOfBinaryTree(root) << endl;
    return 0;
}
```

6) Aim: Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Input: root = [1,2,3,null,5,null,4]

Output: [1,3,4]

Constraints:

The number of nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

Algorithm:

1. **Use Level Order Traversal (BFS)**:
   - Traverse the tree level by level.
   - At each level, store the last node encountered (rightmost node).
2. **Return the collected rightmost nodes**.

Code:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

vector<int> rightSideView(TreeNode* root) {
    if (!root) return {};

    vector<int> result;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        TreeNode* rightmostNode = nullptr;

        for (int i = 0; i < levelSize; ++i) {
```

```
            TreeNode* node = q.front();
            q.pop();
            rightmostNode = node;

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        result.push_back(rightmostNode->val);
    }

    return result;
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(4);

    vector<int> output = rightSideView(root);
    for (int val : output) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

7) Aim: Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

Output: true

Constraints:

The number of nodes in the tree is in the range [0, 5000].

-1000 <= Node.val <= 1000

-1000 <= targetSum <= 1000

Algorithm:

1. **Base Case:**
   - If the tree is empty (root == nullptr), return false.
   - If the node is a leaf (root->left == nullptr && root->right == nullptr), check if targetSum == root->val.
2. **Recursive Case:**
   - Subtract the current node's value from targetSum.
   - Recursively check the left and right subtrees.

Code:

```
#include <iostream>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

bool hasPathSum(TreeNode* root, int targetSum) {
    if (!root) return false;

    targetSum -= root->val;
    if (!root->left && !root->right) return targetSum == 0;

    return hasPathSum(root->left, targetSum) || hasPathSum(root->right,
targetSum);
}

int main() {
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(4);
    root->right = new TreeNode(8);
    root->left->left = new TreeNode(11);
    root->right->left = new TreeNode(13);
    root->right->right = new TreeNode(4);
    root->left->left->left = new TreeNode(7);
    root->left->left->right = new TreeNode(2);
    root->right->right->right = new TreeNode(1);

    int targetSum = 22;
    cout << (hasPathSum(root, targetSum) ? "true" : "false") << endl;
    return 0;
}
```

Lab Module - XII

1) Aim: Given an undirected and disconnected graph G(V, E), containing 'V' vertices and 'E' edges, the information about edges is given using 'GRAPH' matrix, where i-th edge is between GRAPH[i][0] and GRAPH[i][1]. print its DFS traversal.

V is the number of vertices present in graph G and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

Note: The Graph may not be connected i.e there may exist multiple components in a graph.

Constraints:

2 <= V <= 10^3

1 <= E <= (5 * (10^3))

Sample Input 1:

5 4

0 2

0 1

1 2

3 4

Sample Output 1:

2

0 1 2

3 4

Algorithm:

To perform **Depth-First Search (DFS)** on a **disconnected graph**, we need to:

1. **Iterate through all vertices** to ensure we visit every component.
2. **Use a visited array** to track visited nodes.
3. **Run DFS from each unvisited node** to explore its connected component.

Code:

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>

using namespace std;

void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited,
vector<int>& component) {
    visited[node] = true;
    component.push_back(node);

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adj, visited, component);
        }
    }
}

void dfsTraversal(int V, vector<vector<int>>& GRAPH) {
    vector<vector<int>> adj(V);
    for (auto edge : GRAPH) {
        adj[edge[0]].push_back(edge[1]);
        adj[edge[1]].push_back(edge[0]); // Since it's an undirected graph
    }

    // Sort adjacency lists to ensure consistent traversal order
    for (int i = 0; i < V; ++i) {
        sort(adj[i].begin(), adj[i].end());
    }

    vector<bool> visited(V, false);
    vector<vector<int>> components;

    for (int i = 0; i < V; ++i) {
        if (!visited[i]) {
            vector<int> component;
            dfs(i, adj, visited, component);
            components.push_back(component);
        }
    }

    cout << components.size() << endl; // Number of connected components
    for (auto& comp : components) {
        for (int node : comp) {
            cout << node << " ";
        }
        cout << endl;
    }
}
```

```
int main() {
    int V, E;
    cin >> V >> E;
    vector<vector<int>> GRAPH(E, vector<int>(2));

    for (int i = 0; i < E; ++i) {
        cin >> GRAPH[i][0] >> GRAPH[i][1];
    }

    dfsTraversal(V, GRAPH);
}
```

2) Aim: **There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.**

You want to determine if there is a valid path that exists from vertex source to vertex destination. Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise.

Input: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2

Output: true

Constraints:

1 <= n <= 2 * 105

0 <= edges.length <= 2 * 105

edges[i].length == 2

0 <= ui, vi <= n - 1

ui != vi

0 <= source, destination <= n - 1

There are no duplicate edges.

There are no self edges.

Algorithm:

1. **Build an adjacency list** from the given edges.
2. **Use DFS or BFS** to traverse the graph starting from source.
3. **Check if** destination **is reachable**.

Code:

```cpp
#include <iostream>
#include <vector>

using namespace std;

void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited) {
    visited[node] = true;
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adj, visited);
        }
    }
}

bool validPath(int n, vector<vector<int>>& edges, int source, int destination)
{
    vector<vector<int>> adj(n);
    for (auto edge : edges) {
        adj[edge[0]].push_back(edge[1]);
        adj[edge[1]].push_back(edge[0]); // Since it's an undirected graph
    }

    vector<bool> visited(n, false);
    dfs(source, adj, visited);

    return visited[destination];
}

int main() {
    int n, e, source, destination;
    cin >> n >> e;
    vector<vector<int>> edges(e, vector<int>(2));

    for (int i = 0; i < e; ++i) {
        cin >> edges[i][0] >> edges[i][1];
    }
    cin >> source >> destination;

    cout << (validPath(n, edges, source, destination) ? "true" : "false") <<
endl;
}
```

3) Aim: There are n rooms labeled from 0 to n - 1 and all the rooms are locked except for room 0. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of distinct keys in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms. Given an array rooms where rooms[i] is the set of keys that you can obtain if you visited room i, return true if you can visit all the rooms, or false otherwise.

Input: rooms = [[1],[2],[3],[]]

Output: true

Constraints:

n == rooms.length

2 <= n <= 1000

0 <= rooms[i].length <= 1000

1 <= sum(rooms[i].length) <= 3000

0 <= rooms[i][j] < n

All the values of rooms[i] are unique.

Algorithm:

1. **Start from Room 0** and collect all available keys.
2. **Use DFS or BFS** to visit rooms as keys are discovered.
3. **Track visited rooms** to avoid redundant checks.
4. **Return** true **if all rooms are visited**, otherwise return false.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

void dfs(int room, vector<vector<int>>& rooms, vector<bool>& visited) {
    visited[room] = true;
    for (int key : rooms[room]) {
        if (!visited[key]) {
            dfs(key, rooms, visited);
        }
    }
}

bool canVisitAllRooms(vector<vector<int>>& rooms) {
    int n = rooms.size();
    vector<bool> visited(n, false);
```

```
    dfs(0, rooms, visited);

    for (bool v : visited) {
        if (!v) return false;
    }
    return true;
}

int main() {
    int n;
    cin >> n;
    vector<vector<int>> rooms(n);

    for (int i = 0; i < n; ++i) {
        int k;
        cin >> k;
        rooms[i].resize(k);
        for (int j = 0; j < k; ++j) {
            cin >> rooms[i][j];
        }
    }

    cout << (canVisitAllRooms(rooms) ? "true" : "false") << endl;
}
```

4) Aim: You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels. You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

Input: deadends = ["0201","0101","0102","1212","2002"], target = "0202"

Output: 6

Constraints:

1 <= deadends.length <= 500

deadends[i].length == 4

target.length == 4

target will not be in the list deadends.

target and deadends[i] consist of digits only.

Algorithm:

1. **Use BFS to explore possible lock states**:
    - Start from "0000" and generate all possible next states by rotating each wheel forward or backward.
    - If a state is in deadends, ignore it.
    - Track visited states to avoid cycles.
2. **Stop when the target is reached**.
3. **Return the number of moves** required to reach the target.

Code:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>

using namespace std;

vector<string> getNextStates(string current) {
    vector<string> nextStates;
    for (int i = 0; i < 4; ++i) {
        char digit = current[i];
        string up = current, down = current;
        up[i] = (digit == '9') ? '0' : digit + 1;
        down[i] = (digit == '0') ? '9' : digit - 1;
        nextStates.push_back(up);
        nextStates.push_back(down);
    }
    return nextStates;
}

int openLock(vector<string>& deadends, string target) {
    unordered_set<string> dead(deadends.begin(), deadends.end());
    if (dead.count("0000")) return -1;

    queue<pair<string, int>> q;
    unordered_set<string> visited;
    q.push({"0000", 0});
    visited.insert("0000");

    while (!q.empty()) {
        auto [current, moves] = q.front();
        q.pop();
```

```
        if (current == target) return moves;

        for (string next : getNextStates(current)) {
            if (!dead.count(next) && !visited.count(next)) {
                visited.insert(next);
                q.push({next, moves + 1});
            }
        }
    }

    return -1;
}

int main() {
    int n;
    cin >> n;
    vector<string> deadends(n);
    for (int i = 0; i < n; ++i) {
        cin >> deadends[i];
    }
    string target;
    cin >> target;

    cout << openLock(deadends, target) << endl;
}
```

5) Aim: Given an array of non-negative integers arr, you are initially positioned at start index of the array. When you are at index i, you can jump to i + arr[i] or i - arr[i], check if you can reach any index with value 0.

Notice that you can not jump outside of the array at any time.

Input: arr = [4,2,3,0,3,1,2], start = 5

Output: true

Constraints:

1 <= arr.length <= 5 * 104

0 <= arr[i] < arr.length

0 <= start < arr.length

Algorithm:

1. **Model the problem as a graph**:
    - Each index is a node.
    - Valid jumps (i + arr[i] and i - arr[i]) act as edges.
    - The goal is to reach a node with value 0.
2. **Use BFS or DFS**:
    - Track visited indices to prevent infinite loops.
    - Start from start index and explore all reachable indices.
    - If an index with value 0 is found, return true.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

bool dfs(vector<int>& arr, int index, vector<bool>& visited) {
    if (index < 0 || index >= arr.size() || visited[index]) return false;
    if (arr[index] == 0) return true;

    visited[index] = true;
    return dfs(arr, index + arr[index], visited) || dfs(arr, index -
arr[index], visited);
}

bool canReach(vector<int>& arr, int start) {
    vector<bool> visited(arr.size(), false);
    return dfs(arr, start, visited);
}

int main() {
    int n, start;
    cin >> n >> start;
    vector<int> arr(n);

    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    cout << (canReach(arr, start) ? "true" : "false") << endl;
}
```

6) Aim: There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

Input: numCourses = 2, prerequisites = [[1,0]]

Output: true

Constraints:

1 <= numCourses <= 2000

0 <= prerequisites.length <= 5000

prerequisites[i].length == 2

0 <= ai, bi < numCourses

All the pairs prerequisites[i] are unique.

Algorithm:

1. **Model the problem as a directed graph**:
    - Each course is a node.
    - Each prerequisite [a, b] represents a directed edge b → a.
2. **Detect cycles**:
    - If a cycle exists, it is **impossible** to complete all courses.
    - If no cycle exists, a valid ordering exists.

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
    vector<vector<int>> adj(numCourses);
    vector<int> inDegree(numCourses, 0);

    for (auto& pre : prerequisites) {
        adj[pre[1]].push_back(pre[0]);
        inDegree[pre[0]]++;
    }

    queue<int> q;
```

```
    for (int i = 0; i < numCourses; ++i) {
        if (inDegree[i] == 0) q.push(i);
    }

    int count = 0;
    while (!q.empty()) {
        int course = q.front();
        q.pop();
        count++;

        for (int neighbor : adj[course]) {
            if (--inDegree[neighbor] == 0) q.push(neighbor);
        }
    }

    return count == numCourses;
}

int main() {
    int numCourses, n;
    cin >> numCourses >> n;
    vector<vector<int>> prerequisites(n, vector<int>(2));

    for (int i = 0; i < n; ++i) {
        cin >> prerequisites[i][0] >> prerequisites[i][1];
    }

    cout << (canFinish(numCourses, prerequisites) ? "true" : "false") <<
endl;
}
```